University of Alberta

DYNAMIC ADJACENCY LABELLING SCHEMES

by

David Morgan ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Fall 2006

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-23083-1
Our file Notre référence
ISBN: 978-0-494-23083-1

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

As defined by Muller [Muller, Ph.D. thesis, Georgia Tech, 1988] and Kannan, Naor, and Rudich [Kannan et al., SIAM J Disc Math, 1992], an *adjacency labelling scheme* labels a graph such that the adjacency of two vertices can be deduced implicitly from their labels. In general, the labels used in adjacency labelling schemes cannot be tweaked to reflect small changes in the graph.

First studied by Brodal and Fagerberg [Brodal and Fagerberg, LNCS 1663, 1999], a *dynamic adjacency labelling scheme* is an adjacency labelling scheme that requires only small adjustments to the vertex labels when a small change is made to the graph. Motivated by the necessity for further exploration of dynamic adjacency labelling schemes, we introduce the concept of error-detection, discuss metrics for judging the quality of dynamic schemes, and develop error-detecting fully dynamic schemes for several classes of graphs.

Our dynamic scheme for line graphs uses $O(\log n)$ bit labels and updates in $O(e)$ time, where e is the number of edges added to, or deleted from, the line graph. As well, our dynamic scheme for proper interval graphs uses $O(\log n)$ bit labels and handles all operations in $O(n)$ time.

We also develop a $O(r \log n)$ bit/label dynamic adjacency labelling scheme for *r-minoes*, which are graphs with no vertex in more than r maximal cliques. Edge addition and deletion can be handled in $O(r^2\mathbf{D})$ time, vertex addition in $O(r^2e^2)$ time, and vertex deletion in $O(r^2e)$ time, where $\mathbf{D}$ is the maximum degree of the vertices in the original graph and e is the number of edges added to, or deleted from, the original graph.

Similar to this dynamic scheme for r-minoes, we develop a $O(r \log n)$ bit/label dynamic adjacency labelling scheme for *r-bics*, which are graphs with no vertex in more than r maximal bicliques. Edge addition and deletion, as well as vertex deletion, can be handled in $O(r^2\mathbf{B})$ time, and vertex addition in $O(r^2n\mathbf{B})$ time, where $\mathbf{B}$ is the size of the largest biclique in the original graph.

Our dynamic labelling schemes for r-minoes and r-bics lead to $O(r^2n^3)$ time recognition algorithms for both of these classes.

Acknowledgements

First, I would like to acknowledge the support of my wife, Angela, whose confidence in my abilities far exceeds my own. She is my biggest fan.

As well, I would like to thank Dr. Lorna Stewart, my doctoral supervisor, for giving me the freedom to independently pursue my own research interests. Although our research areas are not closely related, her experience offered many insights into my work. Moreover, I would like to thank her for being open to, and supportive of, the idea of finishing my doctoral studies from over 4000 kilometers away.

Finally, I would like to thank NSERC, iCORE, the Killam Scholarship Program, the Alberta Scholarship Program, and the University of Alberta for providing financial support. Without financial support from these programs and organizations, I might never have finished; worse yet, I might never have started ....

Table of Contents

1 Introduction **1**

1.1 Graph terminology 1

1.2 Overview 2

2 Background **6**

2.1 Adjacency labelling schemes 6

2.1.1 Examples 7

2.1.2 Assessing quality 10

2.1.3 Two classes of note 12

2.1.4 Previous work 13

2.1.5 Muller and Kannan et al. 16

2.2 Informative labelling schemes 18

2.2.1 Definition 18

2.2.2 Applications 19

2.2.3 Previous work 20

2.2.4 Dynamization 22

3 Dynamic Schemes **23**

3.1 Dynamic adjacency labelling schemes 23

3.1.1 Definition 23

3.1.2 Example 24

3.1.3 Underlying assumptions 24

3.2 Dynamic informative labelling schemes 25

3.2.1 Definition 25

3.2.2 Assessing quality 26

3.2.3 Graph recognition 29

3.2.4 Previous work 32

3.3 New dynamic schemes 34

4 Line graphs **35**
4.1 Partition isomorphism 35
4.2 The dynamic scheme 38
4.2.1 Vertex labels, marker, and decoder 38
4.2.2 Relabeller 41
4.3 Summary 52

5 *r*-graphs **54**
5.1 The dynamic scheme for *r*-minoes 55
5.1.1 Vertex labels and decoder 55
5.1.2 Relabeller 57
5.2 The dynamic scheme for *r*-bics 67
5.2.1 Vertex labels and decoder 67
5.2.2 Relabeller 70
5.3 Summary 81

6 Proper interval graphs **82**
6.1 Vertex labels, marker, and decoder 84
6.1.1 Relabeller 87
6.2 Summary 107

7 Conclusion **111**

Bibliography **114**

A Definitions **118**

B Computation Models **124**

C Pseudocode **126**
C.1 Line graphs 126
C.1.1 Deleting a vertex 126
C.1.2 Adding a vertex 128
C.1.3 Deleting an edge 140
C.1.4 Adding an edge 154
C.2 *r*-minoes 155
C.2.1 Deleting a vertex 155
C.2.2 Adding a vertex 157
C.2.3 Deleting an edge 159
C.2.4 Adding an edge 160

C.3 r-bics 162
  C.3.1 Deleting a vertex 162
  C.3.2 Adding a vertex 164
  C.3.3 Deleting an edge 167
  C.3.4 Adding an edge 170
C.4 Proper interval graphs 170
  C.4.1 Deleting a vertex 170
  C.4.2 Adding a vertex 172
  C.4.3 Deleting an edge 181
  C.4.4 Adding an edge 184

List of Tables

2.1 Known results on adjacency labelling schemes 14

2.2 Known results on informative labeling schemes for functions other than adjacency 21

4.1 Casewise approach to deletion of an edge (line graphs) 49

4.2 Partition non-isomorphic bases for edge deletion (line graphs) 51

4.3 Partition non-isomorphic bases for edge addition (line graphs) 53

List of Figures

1.1 Illustration of adjacency labelling (interval graphs) 3

2.1 Illustration of adjacency labelling (trees) 7
2.2 Illustration of adjacency labelling (transitive closures of trees) 8
2.3 Illustration of adjacency labelling (line graphs) 9
2.4 Illustration of adjacency labelling (outdegree-k graphs) 10

3.1 Illustration of modification excess and locality 28
3.2 Algorithms for dynamic scheme (graphs of bounded arboricity) 32

4.1 Partition isomorphism 37
4.2 Illustration of dynamic vertex labels (line graphs) 40
4.3 Illustration of edge deletion and addition (line graphs) 42
4.4 Illustration of vertex deletion and addition (line graphs) 42
4.5 DeleteVertex (line graphs) 43
4.6 AddVertex (line graphs) 45
4.7 DeleteEdge (line graphs) 50

5.1 Illustration of dynamic vertex labels (r-minoes) 56
5.2 DeleteVertex (r-minoes) 59
5.3 AddVertex (r-minoes) 62
5.4 DeleteEdge (r-minoes) 65
5.5 AddEdge (r-minoes) 67
5.6 Illustration of dynamic vertex labels (r-bics) 68
5.7 DeleteVertex (r-bics) 71
5.8 AddVertex (r-minoes) 73
5.9 DeleteEdge (r-bics) 78

6.1 An astral triple 82
6.2 Interval representations and blocks 83
6.3 Deleting a vertex I (proper interval graphs) 90

6.4 Merging blocks 91
6.5 LEFTCOMPONENTBLOCKSTRUCTURE 95
6.6 Adding a vertex I (proper interval graphs) 98
6.7 Adding a vertex II (proper interval graphs) 99
6.8 Adding a vertex III (proper interval graphs) 100
6.9 Adding a vertex IV (proper interval graphs) 101
6.10 Adding a vertex V (proper interval graphs) 102
6.11 Adding a vertex VI (proper interval graphs) 102
6.12 Adding a vertex VII (proper interval graphs) 108
6.13 Adding a vertex VIII (proper interval graphs) 108
6.14 Deleting an edge (proper interval graphs) 109
6.15 Adding an edge (proper interval graphs) 110

C.1 Changing the base I (line graphs) 134
C.2 Changing the base II (line graphs) 134
C.3 Changing the base III (line graphs) 136
C.4 Changing the base IV (line graphs) 136
C.5 Changing the base V (line graphs) 137
C.6 Changing the base VI (line graphs) 137
C.7 Deleting an edge I (line graphs) 145
C.8 Deleting an edge II (line graphs) 146
C.9 Deleting an edge III (line graphs) 146
C.10 Deleting an edge IV (line graphs) 147
C.11 Deleting an edge V (line graphs) 148
C.12 Deleting an edge VI (line graphs) 149
C.13 Deleting an edge VII (line graphs) 150
C.14 Deleting an edge VIII (line graphs) 151
C.15 Deleting an edge IX (line graphs) 152
C.16 Deleting an edge X (line graphs) 153
C.17 Deleting an edge XI (line graphs) 155

List of Symbols

The following is a summary of undefined notation used in the thesis.

General notation

$\emptyset$	empty set, empty graph
$\cap$	intersection
$\cup$	union
Σ	summation
$\mathrm{o}(f(n))$	asymptotically less than $f(n)$
$\mathrm{O}(f(n))$	asymptotically less than or equal to $f(n)$
$\Theta(f(n))$	asymptotically equal to $f(n)$
$\Omega(f(n))$	asymptotically greater than or equal to $f(n)$
$\omega(f(n))$	asymptotically greater than $f(n)$
$A \Longrightarrow B$	A implies B
$A \Longleftrightarrow B$	A if and only if B
C_k	cycle on k vertices
$\deg(v)$	degree of vertex v
$\mathrm{dist}_G(v,u)$	distance of vertex v from vertex u in graph G
$\mathrm{dist}_G(v,S)$	distance of vertex v from set of vertices S in graph G
E_G	edge set of graph G
$f(n)$	function of n
$f : X \longrightarrow Y$	a function whose domain is X and range is contained in Y
$G - v$	graph formed by deleting vertex v from graph G
$G + v$	graph formed by adding vertex v to graph G
$G - e$	graph formed by deleting edge e from graph G
$G + e$	graph formed by adding edge e to graph G
$G \Box H$	Cartesian product of graphs G and H
K_k	complete graph on k vertices
$K_{i,j}$	complete bipartite graph (maximal independent set sizes i and j)
$\log x$	logarithm base two of x
$L(G)$	line graph of graph G
$\max\{S\}$	maximum in set S
$\min\{S\}$	minimum in set S
$N(v)$	open neighbourhood of vertex v
$N[v]$	closed neighbourhood of vertex v
$\mathrm{parent}(v)$	parent of vertex v
P_k	path on k vertices
$\mathbb{R}^k$	k-dimensional space over reals

$\lvert S\rvert$	cardinality of set S
uv	an edge uv
V_G	vertex set of graph G
$\lfloor x\rfloor$	floor of x
$\lceil x\rceil$	ceiling of x
$\sqrt{x}$	square root of x
$x!$	factorial of x
$x \in X$	x belongs to set X
$\{x \mid P(x)\}$	set of all elements that satisfy property P
$x \prec y$	x precedes y
$x \preceq y$	x precedes or is y
$x \succ y$	x succeeds y
$x \succeq y$	x succeeds or is y
$X \subset Y$	X is a proper subset of Y
$X \subseteq Y$	X is a subset of Y
$X \supset Y$	X is a proper superset of Y
$X \supseteq Y$	X is a superset of Y
$X \setminus Y$	set X setminus set Y

Algorithm notation

Dequeue(Q)	remove from head of queue Q
Enqueue(Q)	add to tail of queue Q
nil	empty stack or queue
Pop(S)	remove from top of stack S
Push(S)	add to top of stack S
$x \leftarrow y$	assign y to x

Chapter 1

Introduction

1.1 Graph terminology

Let us begin by revisiting some graph terminology. For the definitions of terms used in the thesis, but not defined herein, the reader should consult West [57].

An *undirected graph* $G = (V_G, E_G)$ consists of a *vertex set* V_G, and an *edge set* E_G, where each member of E_G is a subset of V_G having size either one or two. The members of V_G are known as *vertices*, and the members of E_G are known as *edges*, where in particular, if an edge consists of only one vertex, it is called a *loop*.

Given an undirected graph G, two vertices u and v are said to be *adjacent* if the edge $\{u, v\}$ belongs to G; moreover, the vertices u and v are said to be *incident* with the edge $\{u, v\}$, and vice-versa. For simplicity, we will denote the edge $\{u, v\}$ by uv, which is a common practice in graph theoretical literature. Two edges are said to be *adjacent* if they share a common vertex.

The *open neighbourhood* of a vertex v, denoted $N(v)$, is the set of vertices to which v is adjacent. The *closed neighbourhood* of v, denoted $N[v]$, is defined to be $N(v) \cup \{v\}$. When we wish to refer to the closed neighbourhood of a vertex we will do so explicitly; as such, any references to the neighbourhood of a vertex are to its open neighbourhood. The *degree* of a vertex is the cardinality of its open neighbourhood.

A *directed graph* $G = (V_G, E_G)$ consists of a *vertex set* V_G, and an *edge set* E_G, where each member of E_G is an ordered pair of V_G having size either one or two. A directed graph is similar to a graph, however, its edges are ordered pairs; thereby, a direction is imparted to each of its edges. Consequently, the terms defined for undirected graphs, have directed counterparts such as inneighbourhood, outneighbourhood, indegree, and outdegree. A graph, undirected or directed, is *simple* if it contains no loops, and *finite* if its vertex set is finite. Our usage of the term graph will refer to a finite simple undirected graph. When deviating from this usage, we will explicitly state the type of graph under consideration.

A *walk* between two vertices u and v is a sequence of edges that lead from u to v. A walk which visits no vertex twice is known as a *path*. Indeed, the term path is seen more often than walk, as the existence of a walk implies the existence of a path. A maximal set of vertices with a path between each pair of members is called a *component*. A graph is said to be *connected* if the entire vertex set is a component.

Two graphs G_1 and G_2 are said to be *isomorphic* if there exists a bijection $f : V_{G_1} \Longrightarrow V_{G_2}$, for which $uv \in E_{G_1} \Leftrightarrow f(u)f(v) \in E_{G_2}$. We will used the term *labelled graph* to refer to a graph G whose vertex set is $\{1, 2, \ldots, |V_G|\}$. A graph is said to be *unlabelled* if it is not labelled. Two labelled graphs are said to be *distinct* if they are unequal. Two unlabelled graphs are said to be *distinct* if they are not isomorphic.

1.2 Overview

Consider a finite simple undirected graph $G = (V_G, E_G)$ with n vertices and m edges; typically, we represent G using an adjacency matrix, labelling the vertices from 1 to n. These labels serve only to distinguish between the vertices and do not tell us anything about the structure of G. In particular, the adjacency of any pair of vertices is determined from the adjacency matrix, which is usually maintained as a global resource.

What if we could determine the adjacency of two vertices of G in a more local manner? One such way is to use an adjacency list representation which requires $\Theta((m+n)\log n)$ bits to represent a graph. Unfortunately, for dense graphs, an adjacency list representation can require as many as $\Theta(n^2 \log n)$ bits, which is much greater than the $\Theta(n^2)$ bits required by an adjacency matrix representation.

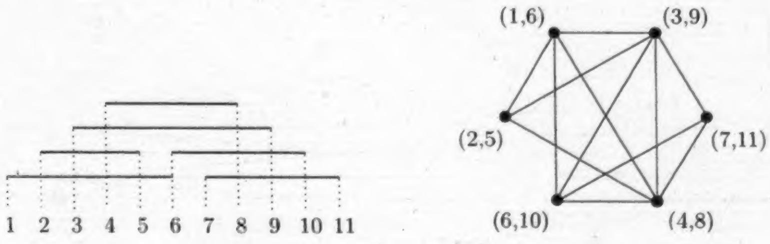
Another approach is to use an adjacency labelling scheme, as defined by Muller [45] and Kannan, Naor, and Rudich [31].

Definition 1.1 *An adjacency labelling scheme of a family $\mathcal{G}$ of finite graphs is a pair of algorithms, (M, D), satisfying the following.*

- *M is a vertex labelling algorithm (marker) whose input is a graph G in $\mathcal{G}$. Note that M need not be deterministic; accordingly, let $\mathcal{M}_G$ be the set of all vertex labellings of V_G which can be assigned by M.*
- *D is a polynomial time deterministic evaluation algorithm (decoder) whose input is a pair of vertex labels. For any graph G in $\mathcal{G}$, and for any labelling M_G generated by M, we require that D be able to correctly determine the adjacency of any pair of vertices of G, using only their labels; we refer to this property by saying that D is adjacency correct.*

In essence, an adjacency labelling scheme is a distributed data structure that allows us to quickly determine the adjacency of two vertices from local information. To date, space-optimal adjacency labelling schemes have been developed for a variety of graph classes, such as bounded arboricity graphs, line graphs, and interval graphs [45, 31].

For example, consider the following adjacency labelling scheme for interval graphs [45]. Recall that a graph is said to be an *interval graph* if each vertex can be represented by an interval of real numbers such that two vertices are adjacent if and only if the corresponding intervals have non-empty intersection. Any such interval representation can be mapped to another interval representation using closed intervals with endpoints in $\{1, \ldots, 2n\}$. The marker labels each vertex with the two endpoints of its associated interval while the decoder determines adjacency in O(1) time by comparing these integers just as it would two intervals. Each label requires $O(\log n)$ bits and the entire labelling uses $O(n \log n)$ bits. An example of an adjacency labelling of an interval graph is given in Figure 1.1.



(a) An interval representation of a graph.

(b) The corresponding labels.

Figure 1.1: An adjacency labelling of an interval graph

Adjacency can be replaced by any function f defined on sets of vertices. In turn, for any set S of vertices on which f is defined, D must output the correct value of f on S using only the labels of the vertices in S. By setting adjacency labelling schemes in the larger context of *informative labelling schemes*, Peleg [47] rejuvenated interest in the idea of space efficient distributed data structures as introduced by Muller [45] and Kannan et al. [31]. To date, informative labelling schemes have been developed for a variety of functions including distance, routing, center of three vertices, ancestor, and nearest common ancestor. In Chapter 2, we offer a comprehensive look at the theory of informative labelling schemes.

In many applications the underlying topology is constantly changing and we desire algorithms which can accommodate these changes. At present, algorithms for finding informative labelling schemes are static; that is, if a graph is changed then the algorithm must devise a labelling of the new graph from scratch. The dynamic version of adjacency labelling schemes was mentioned by Kannan et al. [31], however, they did not consider the problem in detail. At most, the authors suggest that the addition or deletion of a vertex or an edge should require only a "quick" update of the labels in order to obtain an adjacency labelling of the new graph. The first paper to address this dynamic problem is that of Brodal and Fagerberg [10] who develop a dynamic adjacency labelling scheme for graphs of bounded arboricity, providing the graph operations do not cause the arboricity bound to be violated. More recently, the papers of Korman and Peleg [37] and Korman, Peleg, and Rodeh [39] have considered the dynamic problem for trees in the context of distributed computing. Cohen, Kaplan, and Milo [11] consider dynamic ancestor labellings of XML trees with persistent labels; that is, the label of a vertex cannot be changed once it has been assigned. In contrast, our labels can change over time. By not using persistent labels it is possible to reduce label size as we can change the labels as required, or as desired.

As a continuation of the aforementioned works, we further discuss and develop the theory of these dynamic schemes. In Chapter 3, we formally define what is meant by a dynamic informative labelling scheme, as previous literature on this subject has been based exclusively on our intuitive understanding of how static problems are made dynamic. While presenting this formal definition, we introduce the concept of error-detection; that is, the algorithms which relabel the graph should recognize when the modified graph is no longer a member of the family under consideration. In addition to formally defining these schemes, we also demonstrate the connection between error-detection and the graph recognition problem, and identify and discuss the qualities that make a good dynamic scheme.

The latter three chapters of this thesis develop dynamic adjacency labelling schemes for four classes of graphs. Common to the development of dynamic schemes for all these classes is the use of identifiers and circular linked lists to encode information at the vertex level. Specifically, each vertex of a graph G is assigned a unique identifier from $\{1, \dots, |V_G|\}$. For some substructure S of G, such as a maximal clique, the label of a member of S will include the identifier of the next vertex in a circular linked list of vertices in the substructure. In fact, we can incorporate a circular doubly linked list without an increase in the asymptotic size of the vertex label, as we store two identifiers instead of one. By incorporating these circular linked lists we can determine all the vertices in S from a single vertex of S.

Using this distributed representation of S, we can also include additional information about S in the labels of its member vertices. For example, one of the substructures seen in Chapter 5 is a maximal biclique. For each maximal biclique B, we use the circular linked list technique to distribute the representation of B, however, in the label of each member vertex of B to which we identify which part of the bipartition of B the vertex belongs. With this additional information, we are able to develop a decoder for our dynamic scheme.

Not only do we develop a technique to distribute graph substructures across their member vertices, but we also develop a technique to distribute pointers. Consider a pointer P which points from one substructure S_1 to another substructure S_2. First, we select a pointer vertex from S_1, denoted $P(S_1)$. The label of each member of S_1 specifies the identifier of $P(S_1)$, where the label of $P(S_1)$ contains a field which holds the identifier of a vertex in S_2. In turn, the label of that member of S_2 specifies the identifier of $P(S_2)$, so pointers can be followed at will. This technique is used in Chapter 6, where the graph substructures have a linear ordering.

Each of the dynamic adjacency labelling schemes that we develop is fully dynamic, that is, the graph operations allowed are the addition or deletion of a vertex (along with its incident edges), and the addition or deletion of an edge. Moreover, each dynamic scheme is error-detecting. In Chapter 4, we present a dynamic adjacency labelling scheme for line graphs, a class of graphs fundamental in the study of intersection graph theory [8]. Our dynamic scheme for line graphs uses $O(\log n)$ bit labels and updates in $O(e)$ time, where e is the number of edges added to, or deleted from, the line graph. In developing this dynamic scheme, we introduce a new concept known as partition isomorphism, and develop theory on the types of line graphs that can be dynamically altered to produce new line graphs.

In Chapter 5, we present data structures based on maximal cliques and maximal bicliques that give rise to dynamic adjacency labelling schemes for two classes of graphs. Specifically, we develop an $O(r \log n)$ bit/label dynamic adjacency labelling scheme for *r-minoes*, defined by Metelsky and Tyshkevich [44] as the class of graphs with no vertex in more than r maximal cliques. Edge addition and deletion can be handled in $O(r^2\mathbf{D})$ time, vertex addition in $O(r^2e^2)$ time, and vertex deletion in $O(r^2e)$ time, where $\mathbf{D}$ is the maximum degree of the vertices in the original graph and e is the number of edges added to, or deleted from, the original graph.

Similar to this dynamic scheme for r-minoes, we develop a $O(r \log n)$ bit/label dynamic adjacency labelling scheme for *r-bics*, a new class which we define as the graphs with no vertex in more than r maximal bicliques. Edge addition and deletion, as well as vertex deletion, can be handled in $O(r^2\mathbf{B})$ time, and vertex addition in $O(r^2n\mathbf{B})$ time, where $\mathbf{B}$ is the size of the largest biclique in the original graph.

Finally, in Chapter 6, we present a dynamic adjacency labelling scheme for proper interval graphs, a subclass of interval graphs. Our dynamic scheme for proper interval graphs uses $O(\log n)$ bit labels and handles all operations in $O(n)$ time.

Chapter 2

Background

2.1 Adjacency labelling schemes

Recall Definition 1.1, the definition of an adjacency labelling scheme. Allowing sufficiently large labels we can create an adjacency labelling scheme for any family of graphs.

For instance, consider labelling each vertex with a unique "identifier" from $\{1, \ldots, n\}$ (for simplicity, we will refer to vertices by their identifiers), along with its corresponding row of the adjacency matrix [53]. We can determine the adjacency of v_1 and v_2, using only their labels, by looking up the bit corresponding to v_2 in the row of the adjacency matrix found in the label of v_1, or vice versa. Each label requires $\Theta(n)$ bits, the entire labelling requires $\Theta(n^2)$ bits, and adjacency queries require $O(1)$ time (throughout this work we assume a word-level RAM computation model for the marker and decoder, where word sizes are $\Omega(\log n)$; a comparison between this and other common computation models, as well as a discussion on why this model was chosen, can be found in Appendix B).

Another approach is to label each vertex with a unique identifier from $\{1, \ldots, n\}$, along with a list of the identifiers of the vertices to which it is adjacent (an adjacency list of identifiers). We can determine the adjacency of v_1 and v_2, using only their labels, by determining if v_2 is in the adjacency list found in the label of v_1, or vice versa. The label of vertex v requires as many as $\Theta(\deg(v) \log n) \subseteq O(n \log n)$ bits, the entire labelling requires as many as $\Theta((m + n) \log n)$ bits, and adjacency queries require $O(\log n)$ time, provided the adjacency lists are sorted.

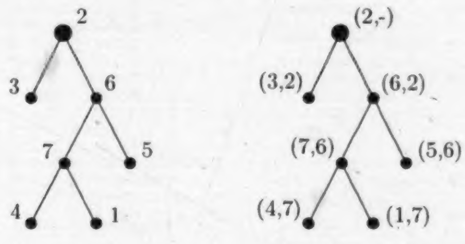
Unfortunately, the adjacency labelling schemes obtained from adjacency matrices and lists are often not space efficient. Many classes of graphs exhibit adjacency labelling schemes that use $O(\log n)$ bit labels, which is a substantial improvement over the $\Theta(n)$ and $O(n \log n)$ bit labels offered by adjacency matrices and adjacency lists, respectively.

2.1.1 Examples

For many graph classes, the defining properties of the class often determine an adjacency labelling scheme. We now present adjacency labelling schemes for a variety of classes, noting that, when we refer to a particular graph class we mean those graphs which are unlabelled; that is, isomorphic copies are not distinct. When we wish to refer to a class of labelled graphs we will do so explicitly.

Trees

Perhaps the simplest adjacency labelling scheme is the following scheme on trees [31]. Consider a tree T on n vertices. The marker assigns an arbitrary root, gives each vertex an arbitrary but unique identifier from $\{1, \ldots, n\}$, then assigns to each vertex v of T the label $(v, \text{parent}(v))$. Each label uses $\mathrm{O}(\log n)$ bits and the marker takes $\Theta(n)$ time to label the graph. The decoder determines the adjacency of two vertices v_1 and v_2 in $\mathrm{O}(1)$ time, using only their labels, by checking if $v_1 = \text{parent}(v_2)$ or $v_2 = \text{parent}(v_1)$. An example of an adjacency labelling of a tree is presented in Figure 2.1.



(a) Identifiers of T.

(b) Labels of T.

Figure 2.1: An adjacency labelling of a tree

Transitive closures of rooted trees

The class of graphs which are transitive closures of rooted trees also has an adjacency labelling scheme [31]. The *transitive closure* T' of a rooted tree T is the graph defined by $V_{T'} = V_T$ and $E_{T'} = \{\{u, v\} | \text{there is a directed path from } u \text{ to } v \text{ in } T\}$.

Consider the transitive closure T' of a rooted tree T on n vertices. Observe that the vertices of T' are exactly the vertices of T, so we may refer to them interchangeably. Let the set of descendants of a vertex v in T be denoted $D(v)$. The marker assigns each vertex a unique identifier from $\{1, \ldots, n\}$ by traversing the tree in postorder, then assigns to each vertex v of T' the label $\left[\min\limits_{w \in D(v)} w, v\right]$. Each label uses $\mathrm{O}(\log n)$ bits and the marker takes $\Theta(n)$ time to label the graph. The decoder determines the adjacency of two vertices v_1 and

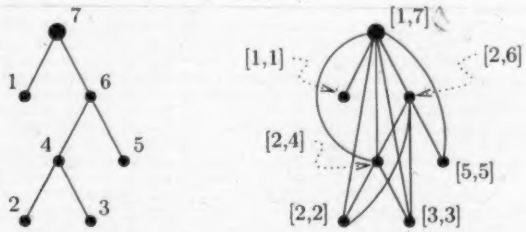
v_2 in O(1) time using only their labels by checking if

$$\min_{w \in D(v_1)} w \leq v_2 \leq v_1$$

or

$$\min_{w \in D(v_2)} w \leq v_1 \leq v_2.$$

An example of an adjacency labelling of a transitive closure of a tree is given in Figure 2.2.



(a) Identifiers of T

(b) Labels of T'

Figure 2.2: An adjacency labelling of the transitive closure of a tree

Line graphs

The class of line graphs also has an adjacency labelling scheme. The definition of a line graph is as follows [8].

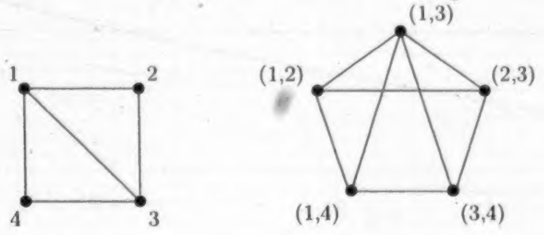
Definition 2.1 *Given a graph $G = (V_G, E_G)$, its line graph is the graph $L(G) = (E_G, E_{L(G)})$ for which $\{u, v\} \in E_{L(G)}$ if and only if u and v are adjacent edges in G.*

We observe that by adding isolated vertices to G we can obtain infinitely many graphs which give rise to the same line graph. As such, if a graph G has no isolated vertices we will refer to it as a base of $L(G)$. Whitney [58] has shown that every connected line graph has a unique base, up to isomorphism, except for K_3 which has two bases, namely, K_3 and $K_{1,3}$.

Consider a line graph $L(G)$, with base graph G. To each vertex in G the marker assigns an arbitrary but unique identifier from $\{1, \ldots, |V_G|\}$, then assigns to each vertex v in $L(G)$ the label (e_1, e_2), where e_1 and e_2 are the identifiers of the endpoints of the edge of G corresponding to v. Since G has no isolated vertices, $|V_G| \leq 2|E_G| = 2|V_{L(G)}|$, so each label uses $O(\log |V_G|) = O(\log |V_{L(G)}|)$ bits.

We assume that the marker is input with G, the base graph, as well as the correspondence between edges in G and vertices in $L(G)$; therefore, the marker can generate an adjacency labelling in $\Theta(n)$ time. If the marker is input only with the structure of $L(G)$, perhaps in the form of an adjacency matrix, then it must use an algorithm like that of Lehot [40] or Roussopoulos [50], which determines G from $L(G)$. Each of these algorithms has running time $\Theta(m+n)$, thereby resulting in a running time of $\Theta(m+n)$ for the marker.

The decoder can determine the adjacency of two vertices, with labels (ep_0, ep_1) and (ep_2, ep_3), in O(1) time by checking if $\{ep_0, ep_1\} \cap \{ep_2, ep_3\} = \emptyset$. An example of an adjacency labelling of a line graph is given in Figure 2.3.



(a) G. (b) Labels of $L(G)$.

Figure 2.3: An adjacency labelling of a line graph

Outdegree-k graphs

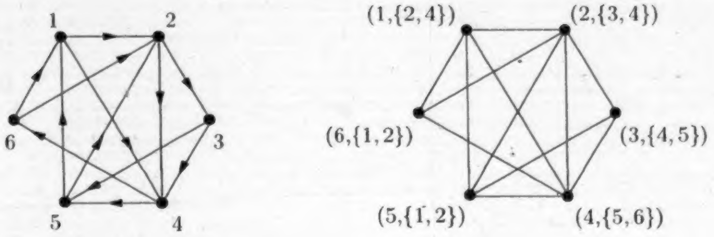
The class of outdegree-k graphs also has an adjacency labelling scheme for any fixed constant k. An outdegree-k graph is defined as follows [31].

Definition 2.2 *An undirected graph is said to be an outdegree-k graph if its edges can be directed such that no vertex has outdegree greater than k. We will call such an orientation an outdegree-k orientation.*

For any fixed constant k, consider an outdegree-k graph G on n vertices and m edges, subject to some outdegree-k orientation $\mathcal{O}$. To each vertex the marker assigns an arbitrary but unique identifier from $\{1, \ldots, n\}$, then assigns to each vertex v the label $(v, \mathcal{O}_v)$, where $\mathcal{O}_v$ denotes the set of identifiers of the outneighbours of v. Each label uses $O(\log n)$ bits.

We assume that the marker knows the orientation $\mathcal{O}$, so it can generate this labelling in $\Theta(n)$ time. If the marker is only input with G, perhaps as an adjacency matrix, then it must use an algorithm like that of Gabow and Westermann [18] which determines $\mathcal{O}$ from G. This algorithm has a running time of $\Theta(kn\sqrt{m + kn\log n})$, resulting in a $\Theta(kn\sqrt{m + kn\log n})$ running time for the marker.

The decoder can determine the adjacency of two vertices, v_1 and v_2, in O(1) time using only their labels by checking if $v_1 \in \mathcal{O}_{v_2}$ or $v_2 \in \mathcal{O}_{v_1}$. An example of an adjacency labelling of an outdegree-2 graph is given in Figure 2.4.



(a) Outdegree-2 orientation and identifiers of G.

(b) Labels of G.

Figure 2.4: An adjacency labelling of an outdegree-2 graph

Subclasses, superclasses, and co-classes

Observe that an adjacency labelling scheme for a class $\mathcal{G}$ is also an adjacency labelling scheme for any subclass of $\mathcal{G}$; the marker and decoder remain the same, but the inputs to the marker are restricted to members of the subclass. Since any tree is an outdegree-1 graph (arbitrarily assign a root and direct each edge toward the parent vertex), the family of trees has an adjacency labelling scheme by virtue of it being a subclass of the outdegree-1 graphs. In fact, the adjacency labelling scheme presented for trees earlier in this section is exactly the restriction of the above scheme for outdegree-1 graphs to trees.

The consequences of the contrapositive of this implication should also be considered. Specifically, if a family of graphs $\mathcal{G}$ does not exhibit an adjacency labelling scheme with property P (for example, label size), then that is also true for any superclass $\mathcal{H}$ of $\mathcal{G}$. If $\mathcal{H}$ did have such a scheme, then the restriction of that scheme to graphs in $\mathcal{G}$ would be an adjacency labelling scheme of $\mathcal{G}$ exhibiting property P.

It should also be observed that if a family $\mathcal{G}$ has an adjacency labelling scheme then so does co-$\mathcal{G}$, the class of graphs whose complements belong to $\mathcal{G}$. Specifically the marker remains the same, but the decoder is the "opposite" decoder; that is, the decoder for co-$\mathcal{G}$ determines that two vertices are adjacent if and only if the decoder for $\mathcal{G}$ determines that the vertices are not adjacent.

2.1.2 Assessing quality

Consider an adjacency labelling scheme (M, D). There are three ways to assess the quality of the scheme.

1. The running time of M.

2. The running time of D.

3. The labels generated by M.

Trivially, the better the running time of M and D, the better the adjacency labelling scheme. Given that M must assign n vertex labels, the running time of any marker is $\Omega(n)$. As we have seen in Section 2.1.1 with line graphs and outdegree-k graphs, the running time of M depends on the representation of the graph provided to M.

Of greater interest are the labels generated by M. We define an adjacency labelling scheme of a family of graphs $\mathcal{G}$ to be *space-optimal* if there is no other adjacency labelling scheme for $\mathcal{G}$ that uses asymptotically fewer total bits. Since vertex adjacency uniquely defines a graph, an adjacency labelling scheme of a family of graphs provides a unique representation for each of the members of the family. Therefore, the total number of bits required by an adjacency labelling scheme is at least the number of bits required to represent all of the members uniquely; in particular, a family of graphs with $2^{\Theta(\phi(n))}$ members on n vertices requires a labelling that uses $\Omega(\phi(n))$ bits (in total, not per vertex) in order to uniquely represent each of the members on n vertices [53]. We will say that an adjacency labelling scheme of a family of size $2^{\Theta(\phi(n))}$ that uses $\Theta(\phi(n))$ total bits is *strongly space-optimal*. For simplicity, we will occasionally refer to a family with $2^{f(n)}$ members on n vertices as having size $2^{f(n)}$.

For example, there are $2^{\Theta(n \log n)}$ interval graphs on n vertices [22], so any adjacency labelling scheme for interval graphs requires $\Omega(n \log n)$ total bits. Therefore, the adjacency labelling scheme presented for interval graphs in Chapter 1 is strongly space-optimal. In contrast, the adjacency labelling schemes created using adjacency matrices and adjacency lists, which use $\Theta(n^2)$ and $\Theta(n^2 \log n)$ total bits, respectively, are not space-optimal for interval graphs.

Although there are many asymptotic counting results for labelled graph classes, the same cannot be said for unlabelled graphs. For instance, we know that there are $2^{\Theta(n \log n)}$ labelled line graphs (hence, there are $2^{O(n \log n)}$ unlabelled line graphs); however, we do not know that there are $2^{\Theta(n \log n)}$ unlabelled line graphs. As such, we cannot say that Muller's adjacency labelling scheme for line graphs, presented in Section 2.1.1, is space-optimal for unlabelled line graphs. As well, consider that there are $2^{O}(n)$ unlabelled trees (consider a binary string encoding of a depth first traversal from an arbitrary root, where 1 denotes going down the tree and 0 denotes moving back up) [52]. As such, the adjacency labelling scheme for trees presented in Section 2.1.1 is not strongly space-optimal for unlabelled line graphs; however, it may be space-optimal.

Along with space-optimality, we are also interested in the property of balance, that is, we want the labels to be of roughly equal size. Specifically, if an adjacency labelling scheme uses $\Theta(\phi(n))$ total bits then we would like each vertex label to use $O(\frac{\phi(n)}{n})$ bits. For example, the adjacency labelling scheme for interval graphs presented in Chapter 1 is balanced. Likewise, the adjacency labelling scheme based on adjacency matrices is balanced for any class of graphs. In contrast, the adjacency labelling scheme created from adjacency lists is not balanced for certain classes of graphs. Specifically, the adjacency list scheme is not balanced for interval graphs as this family contains the complete bipartite graph $K_{1,n-1}$. This graph would have one vertex with a $\Theta(n \log n)$ bit label, while the entire labelling would require $\Theta(n \log n)$ bits in total.

Observe that the scheme devised from adjacency matrices is a balanced strongly space-optimal adjacency labelling scheme (also referred to as a generalized implicit representation by Spinrad [53]) for any family of size $2^{\Theta(n^2)}$. Such families include bipartite graphs, chordal graphs, and the class of all graphs [26, 45]. As such, we are really only interested in whether classes of size $2^{o(n^2)}$ have strongly space-optimal adjacency labelling schemes.

Of the classes of size $2^{o(n^2)}$, balanced strongly space-optimal adjacency labelling schemes have only been found for families of size $2^{\Theta(n \log n)}$. Spinrad [53], presents a space-optimal representation scheme for chordal bipartite graphs, which have $2^{O(n \log^2 n)}$ members on n vertices; however, adjacency testing cannot be performed locally.

Exactly what criteria should be used to assess an adjacency labelling scheme is dependent on the application. It is possible that one could not tolerate increasing the running of time of M to create smaller labels that allow D to run faster. Commonly, however, applications that involve millions of nodes demand that the focus be on the size of the labels generated by M.

2.1.3 Two classes of note with regard to strong space-optimality

Let us now examine two classes of interest with respect to strongly space-optimal adjacency labelling schemes; specifically, k-sparse graphs, a class that does not have a balanced strongly space optimal adjacency labelling scheme, and k-dot product graphs, a class which remains open with respect to the existence of a balanced strongly space-optimal adjacency labelling scheme.

k-sparse graphs

The class of k-sparse graphs is defined as follows [45].

Definition 2.3 *A graph on n vertices is k-sparse if it has at most kn edges.*

Using a proof by contradiction, Muller [45] showed that, for any constant k, the class of k-sparse graphs does not have a balanced strongly space-optimal adjacency labelling scheme. We present a proof similar to that of Muller.

For any constant k, an adjacency list representation uses $O(n \log n)$ bits to represent a labelled k-sparse graph on n vertices; thereby, there are $2^{O(n \log n)}$ labelled k-sparse graphs on n vertices. As such, there are $2^{O(n \log n)}$ unlabelled k-sparse graphs on n vertices. Assume that the class of k-sparse graphs has a balanced strongly space-optimal adjacency labelling scheme, that is, a scheme that uses $O(\log n)$ bits per vertex. Now take any graph, G, on n vertices and add n^2 vertices to it to make it k-sparse. By our assumption, this graph has an adjacency labelling that uses $O(\log(n^2 + n)) = O(\log(n))$ bits per vertex. By restricting any such labelling to G, we obtain an adjacency labelling scheme of the class of all graphs that uses $O(n \log n)$ bits in total. Therefore, there are $2^{O(n \log n)}$ graphs on n vertices, which is a contradiction.

Whether or not k-sparse graphs have a strongly space-optimal adjacency labelling scheme is unknown.

k-dot product graphs

Fiduccia, Scheinerman, Trenk, and Zito [17] define a k-dot product graph as follows.

Definition 2.4 *A graph G is a k-dot product graph if there is a function $f : V_G \longrightarrow \mathbb{R}^k$ such that $f(v_1) \cdot f(v_2) \geq 1$ if and only if v_1 and v_2 are adjacent, where $\cdot$ is the standard inner product of two vectors.*

For any constant k, the class of k-dot product graphs remains open with regards to having an adjacency labelling scheme that uses $O(\log n)$ bits per vertex, even though there are $2^{\Theta(n \log n)}$ members on n vertices [17]. The dot product representation itself is almost such an adjacency labelling scheme, however, there is no upper bound on the number of bits required to represent the members of $\mathbb{R}$. In the same work in which Fiduccia et al. define dot product graphs, they show that the function which defines the dot product representation can actually be restricted to $\mathbb{Q}$, the set of rationals; however the same problem of unbounded representation still exists for $\mathbb{Q}$. What is needed to achieve an adjacency labelling scheme with $O(\log n)$ bits per vertex for this class is a mapping $f : V_G \longrightarrow S^k$, where S is some set whose members can be represented using $O(\log n)$ bits. In their paper, Fiduccia et al. state that they believe that no such set S can be found.

2.1.4 Previous work

Table 2.1 presents known results on adjacency labellings schemes for a variety of graph classes. To assist the reader, definitions of these classes can be found in Appendix A.

In some cases, the upper bound on the size of a class comes from the existence of the adjacency labelling scheme of a particular size. As we have previously discussed in Section 2.1.2, a class exhibiting an adjacency labelling that uses $O(\phi(n))$ bits in total has size $2^{O(\phi(n))}$. For example, there exists an adjacency labelling scheme for outerplanar graphs that uses $O(n \log n)$ bits [45]; therefore, there are $2^{O(n \log n)}$ outerplanar graphs on n vertices.

In other cases, the lower bound on the size of a class is due to the fact that it contains, or is a co-class of, another class whose size we know. For instance, the class of C_3-free graphs contains the class of bipartite graphs, which have size $2^{\Theta(n^2)}$ [45]; therefore, there are $2^{\Omega(n^2)}$ C_3-free graphs on n vertices. Moreover, we know that the class of all graphs, which we refer to as general graphs, has size $2^{\Theta(n^2)}$; therefore, there are $2^{O(n^2)}$ C_3-free graphs on n vertices. In turn, we know that there are $2^{\Theta(n^2)}$ C_3-free graphs on n vertices. Similarly, the class of cobipartite graphs is the co-class of the bipartite graphs; therefore, there are $2^{\Theta(n^2)}$ cobipartite graphs on n vertices. A good resource on classes of size $2^{\Theta(n^2)}$ is Chapter 8 of *Efficient Graph Representations* by Spinrad [53].

Often an adjacency labelling scheme will result from its containment in another class, as per our discussion of subclasses in Section 2.1.1. For example, the class of proper interval graphs have a balanced adjacency labelling scheme using $O(n \log n)$ bits because they are contained in the class of interval graphs which have a balanced adjacency labelling scheme using $O(n \log n)$ bits.

Table 2.1: Known results on adjacency labelling schemes [†number of unlabelled members on n vertices; ‡total bits required in the scheme using asymptotically fewest total bits (the scheme is balanced unless otherwise noted); $^\spadesuit k$ is bounded by a constant; $^\clubsuit$indicates that the scheme is the "default" scheme obtained from adjacency matrices [53]]

Class	**Size**†	**Scheme**‡	**Comments**
General graphs	$2^{\Theta(n^2)}$	$\Theta(n^2)^\clubsuit$	
C_3-free	$2^{\Theta(n^2)}$ [45]	$\Theta(n^2)^\clubsuit$	superclass of bipartite
C_3,C_4-free	$2^{\Omega(n \log n)}$ [53]	$\Theta(n^2)^\clubsuit$	
C_3,$K_{1,3}$-free	$2^{O(n\sqrt{n})}$	$\Theta(n \log n)$ [53]	
K_5-free	$2^{\Theta(n^2)}$ [45]	$\Theta(n^2)^\clubsuit$	superclass of bipartite
$K_{1,3}$-free	$2^{\Theta(n^2)}$ [45]	$\Theta(n^2)^\clubsuit$	superclass of cobipartite
$K_{3,3}$-free	$2^{\Theta(n^2)}$ [45]	$\Theta(n^2)^\clubsuit$	superclass of cobipartite
P_4-free posets	$2^{O(n \log n)}$ [53]	$O(n^2)^\clubsuit$	no $O(n \log n)$ balanced scheme [53]
Almost tree(k)$^\spadesuit$	$2^{O(n \log n)}$	$\Theta(n \log n)$ [45]	
Arboricity-$k^\spadesuit$	$2^{O(n \log n)}$	$\Theta(n \log n)$ [31]	subclass of outdegree-k
Asteroidal triple free	$2^{\Theta(n^2)}$ [8]	$\Theta(n^2)^\clubsuit$	superclass of cobipartite
Autographs	$2^{\Theta(n^2)}$ [45]	$\Theta(n^2)^\clubsuit$	
Bandwidth-$k^\spadesuit$	$2^{O(n \log n)}$	$\Theta(n \log n)$ [45]	subclass of degree-k
Bipartite	$2^{\Theta(n^2)}$ [45]	$\Theta(n^2)^\clubsuit$	
Boxicity-$k^\spadesuit$	$2^{O(n \log n)}$	$\Theta(n \log n)$ [45]	
Chain graphs	$2^{O(n \log n)}$	$\Theta(n \log n)$ [53]	
Chordal	$2^{\Theta(n^2)}$ [45]	$\Theta(n^2)^\clubsuit$	superclass of split
Chordal comparability	$2^{O(n \log n)}$	$\Theta(n \log n)$ [42]	

continued on next page ...

… continued from previous page

Class	**Size**†	**Scheme**‡	**Comments**
Circle	$2^{O(n \log n)}$	$\Theta(n \log n)$ [31, 45]	
Circular arc	$2^{O(n \log n)}$	$\Theta(n \log n)$ [31, 45]	
Cobipartite	$2^{\Theta(n^2)}$ [45]	$\Theta(n^2)^{\clubsuit}$	complements of bipartite graphs
Cographs	$2^{O(n \log n)}$	$\Theta(n \log n)$ [31, 45]	subclass of permutation
Comparability	$2^{\Theta(n^2)}$ [45]	$\Theta(n^2)^{\clubsuit}$	superclass of bipartite
Containment graphs of paths in a tree	$2^{O(n \log n)}$	$\Theta(n \log n)$ [53]	
Convex bipartite	$2^{O(n \log n)}$	$\Theta(n \log n)$ [45]	
Cycles	$2^{O(n \log n)}$	$\Theta(n \log n)$ [45]	
k-decomposable$^{\spadesuit}$	$2^{O(n \log n)}$	$\Theta(n \log n)$ [31]	
Degree-$k^{\spadesuit}$	$2^{O(n \log n)}$	$\Theta(n \log n)$ [31, 45]	subclass of outdegree-k, hereditary degree-k
Disk intersection	$2^{O(n \log n)}$ [53]	$\Theta(n^2)^{\clubsuit}$	
k-dot product$^{\spadesuit}$	$2^{\Theta(n \log n)}$ [17]	$\Theta(n^2)^{\clubsuit}$	
EPT graphs	$2^{O(n \log n)}$	$\Theta(n \log n)$ [53]	
Forest	$2^{O(n \log n)}$	$\Theta(n \log n)$ [31, 45]	
Genus-$k^{\spadesuit}$	$2^{O(n \log n)}$	$\Theta(n \log n)$ [31]	subclass of arboricity-k
Hereditary degree-$k^{\spadesuit}$	$2^{O(n \log n)}$	$\Theta(n \log n)$ [45]	subclass of outdegree-k
k-interval$^{\spadesuit}$	$2^{O(n \log n)}$	$\Theta(n \log n)$ [31]	
Interval	$2^{\Theta(n \log n)}$ [22]	$\Theta(n \log n)$ [31, 45]	subclass of circular arc
Line graphs	$2^{O(n \log n)}$	$\Theta(n \log n)$ [45]	
Outdegree-$k^{\spadesuit}$	$2^{O(n \log n)}$	$\Theta(n \log n)$ [45]	
Outerplanar	$2^{O(n \log n)}$	$\Theta(n \log n)$ [45]	subclass of boxicity-2
Permutation	$2^{O(n \log n)}$	$\Theta(n \log n)$ [31, 45]	subclass of circle
Planar	$2^{O(n \log n)}$	$\Theta(n \log n)$ [31, 45]	subclass of outdegree-k, boxicity-3
Posets (dimension k)$^{\spadesuit}$	$2^{O(n \log n)}$	$\Theta(n \log n)$ [45]	
Proper interval	$2^{\Theta(n)}$[25]	$\Theta(n \log n)$ [45]	subclass of interval
k-sparse$^{\spadesuit}$	$2^{O(n \log n)}$ [53]	$\Theta(n^2)^{\clubsuit}$	no $O(n \log n)$ balanced scheme [53]
Threshold	$2^{O(n \log n)}$	$\Theta(n \log n)$ [45]	subclass of interval
Threshold tolerance	$2^{O(n \log n)}$	$\Theta(n \log n)$ [45]	
Transitive closures of rooted trees	$2^{O(n \log n)}$	$\Theta(n \log n)$ [31]	
Trees	$2^{O(n)}$[52]	$\Theta(n \log n)$ [31, 45]	subclass of forest
Split	$2^{\Theta(n^2)}$ [45]	$\Theta(n^2)^{\clubsuit}$	
Total	$2^{O(n^2)}$	$O(n^2)^{\clubsuit}$	no $O(n \log n)$ balanced scheme [53]
Uniformly k-sparse	$2^{O(n \log n)}$ [53]	$\Theta(n^2)^{\clubsuit}$	

2.1.5 Muller and Kannan et al.

The seminal works of both Muller [45] and Kannan et al. [31] independently introduce a narrower version of adjacency labelling schemes called implicit representations, a term suggested in the title of the work by Kannan et al. [31]. We present the definition of Kannan et al., rather than Muller's, which is built from several smaller definitions.

Definition 2.5 (Kannan et al.) *A family $\mathcal{F}$ of finite graphs has an implicit representation if there is a polynomial time Turing machine T and a function l which labels the vertices of each graph G in $\mathcal{F}$ with distinct labels of* $\mathrm{O}(\log n)$ *bits, n being the number of vertices of G, such that, given two vertex labels of a graph G in $\mathcal{F}$, T will correctly decide adjacency of the corresponding vertices in G.*

Immediately, we notice that this definition uses Turing machines, whereas Definition 1.1 refers to marker and decoder algorithms. Essentially, the Turing machine T employed in Definition 2.5 serves as both the marker, M, and the decoder, D.

As well, we note that Definition 1.1, unlike Definition 2.5, does not require that the vertex labels be distinct. Fortunately, any marker can be modified to obtain unique vertex labels by assigning each vertex an arbitrary but unique identifier from $\{1, \ldots, n\}$ (as is commonly seen in Section 2.1.1) and appending it to the vertex label; this identifier adds a term of $\log n$ to the number of bits required by a vertex label. For any class of size $2^{\Omega(n \log n)}$, the addition of this identifier will not increase the asymptotic size of the vertex labels, as the distinct label requirement necessitates that each label be of size $\Omega(\log n)$. However, for classes with fewer members, this requirement might prevent us from developing strongly space-optimal adjacency labelling schemes. For this reason, the distinct label requirement is omitted from Definition 1.1.

We should note that there is a small but important difference between the definitions found in Muller [45] and Kannan et al. [31]. Specifically, Muller only requires the Turing machine to halt, not to be polynomial, as required by Kannan et al. Since the inputs to the Turing machine are of size $\mathrm{O}(\log n)$, the use of a polynomial Turing machine guarantees that adjacency testing can be performed in polylogarithmic time. In keeping with Kannan et al., the polynomial time requirement has been included in Definition 1.1. That being said, there are no known classes of graphs which "obtain" adjacency labelling schemes when the polynomial time requirement is dropped [53].

Intersection classes and intersection number

In his doctoral thesis, Muller [45] observes that there is an implicit representation for any intersection class in which the vertices of graph can be represented by constant size subsets of $\{1, \ldots, n^k\}$, provided k is a constant. We note that such an implicit representation cannot be constructed without knowing the intersection representation, or at least how to obtain the intersection representation. It is conceivable that such a representation exists, but we do not know how to determine it.

To illustrate Muller's observation on intersection classes, consider the line graphs of simple hypergraphs as defined below [8].

Definition 2.6 *A hypergraph $H = (V, \mathcal{E})$ consists of a set of vertices V, and a set of hyperedges $\mathcal{E}$, which are non-empty subsets of V. A hypergraph is said to be simple if no edge is properly contained within another. The rank of H is the value $\max_{e \in \mathcal{E}}\{|e|\}$.*

Definition 2.7 *The line graph of a hypergraph $H = (V, \mathcal{E})$ is the graph $L(H) = (\mathcal{E}, E)$ for which $ee' \in E$ if and only if $e \neq e'$ and $e \cap e' \neq \emptyset$.*

Consider a line graph of a hypergraph with rank at most k, where k is a constant. If there are n vertices in the line graph, then there are n hyperedges in the hypergraph. Since the rank of the hypergraph is bounded above by k, there are at most kn vertices in the hypergraph. Therefore, the line graph of the hypergraph can be represented by the intersection of subsets of $\{1, \ldots, kn\}$ of size at most k. By Muller's result, this class of graphs has an implicit representation. Unfortunately, this representation is not easily obtained; specifically, the only method known for obtaining such a representation is based upon work that appears in Chapter 5.

Representing the vertices by subsets of $\{1, \ldots, n^k\}$ is equivalent to saying that the intersection number of these graphs is bounded above by n^k. In fact, the intersection number of any graph is in $O(n^2)$. Given any graph G on n vertices, we can uniquely label each edge with a number in $\{1, \ldots, \binom{n}{2}\}$. Each vertex is then represented by the set of labels of edges incident with the vertex. Since the class of all graphs does not have an implicit representation, the condition that the vertices be represented by constant size subsets is critical to Muller's observation.

When we consider Muller's result in the more general context of adjacency labelling schemes, we observe that any intersection class in which the vertices of a graph of order n can be represented by $O(\lambda)$-subsets of $\{1, \ldots, \phi\}$ offers an adjacency labelling scheme with labels using $O(\lambda \log \phi)$ bits. Providing $\lambda \log \phi \in o(n)$, which is to say that $\phi^\lambda \in 2^{o(n)}$, then we enjoy a savings over the "default" adjacency labelling scheme obtained using adjacency matrices.

As mentioned by Muller, these observations on intersection classes also apply if adjacency is determined by any other binary relation such as containment, overlap, or order. For instance, the adjacency labelling scheme for interval graphs presented in Chapter 1 is not based so much on intersection of intervals on the real line; rather, it is really based upon 2-subsets of $\{1, \ldots, 2n\}$ using the binary relation, b, given by

$$b(\{a,b\},\{c,d\}) = \begin{cases} 0 & \text{if } a,b \leq c,d \text{ or } a,b \geq c,d, \\ 1 & \text{otherwise.} \end{cases}$$

Universal graphs

In both of the works of Muller [45] and Kannan et al. [31], implicit representations were shown to have a close relationship to vertex induced universal graphs. These graphs are defined as follows [31].

Definition 2.8 *G is a vertex induced universal graph of a set of graphs S if all members of S are vertex induced subgraphs of G.*

Consider a class of graphs $\mathcal{C}$, and let $\mathcal{C}_n$ denote the set of members of $\mathcal{C}$ having at most n vertices. It has been shown by both Muller and Kannan et al. that if $\mathcal{C}$ has an implicit representation, then, for some constant k, $\mathcal{C}_n$ has a vertex induced universal graph on $O(n^k)$ vertices which can be constructed in polynomial time. Letting T be the Turing machine used in an implicit representation for $\mathcal{C}$, we construct such a universal graph U as follows. In the implicit representation of $\mathcal{C}$, the members of $\mathcal{C}_n$ have labels of length $c \log n$, for some constant c. The vertices of U correspond to each of the $2^{c \log n} = n^c$ bit vectors of length $c \log n$. Two vertices in U, represented by bit vectors b_1 and b_2, are adjacent if and only if $T(b_1, b_2) = 1$.

2.2 Informative labelling schemes

2.2.1 Definition

As mentioned in Chapter 1, adjacency can be replaced by any function f that acts on sets of vertices. We present the definition of an f-labelling scheme as introduced by Peleg [47].

Definition 2.9 *Consider a function $f(S,G)$ defined on sets of vertices S of fixed but arbitrary finite graphs G. An informative f-labelling scheme of a family $\mathcal{G}$ of finite graphs is a pair (M,D) defined as follows.*

- *M is a vertex labelling algorithm (marker) whose input is a graph G in $\mathcal{G}$. Note that M need not be deterministic; accordingly, let $\mathcal{M}_G$ be the set of all vertex labellings of V_G which can be assigned by M.*
- *D is a polynomial time deterministic evaluation algorithm (decoder) whose input is a set of vertex labels. Given any labelling L_G of V_G, let $L_{S,G}$ denote the subset of these labels corresponding to a subset S of V_G. For any graph G in $\mathcal{G}$, we define L_G to be (D,f)-correct if $D(L_{S,G}) = f(S,G)$ for every subset S of V_G on which f is defined.*

Given this definition, we require that M_G be (D, f)-correct for all G in $\mathcal{G}$ and for all M_G in $\mathcal{M}_G$. Note that D is a function of the labels only.

For any such function f, an f-labelling scheme is said to be an informative labelling scheme.

To illustrate this definition, consider the following ancestor labelling for rooted trees. To the root the marker assigns the bit string '0' as its label. For any vertex v, with children $v_1, \ldots, v_k$, the marker assigns to each child a unique member of a prefix-free k-set of strings (by prefix-free we mean that no string is a prefix of another; for instance, '110' is a prefix of '1101', but not of '111'. One such trivial set is $\{0, 10, 110, \ldots\}$). The marker then assigns to each vertex the label consisting of its string from the prefix-free set concatenated to the end of the label of its parent. The decoder can determine if a vertex v_1 is an ancestor of v_2, using only their labels, by checking if the label of v_1 is a prefix of the label of v_2.

Just as with adjacency labelling schemes, we judge the quality of an informative labelling scheme according to the running time of M and D, as well as the labels generated by M. The notions of space-optimality and balance still apply; however, the same cannot be said for strong space-optimality. Fundamental to the concept of strong space-optimality is the fact that adjacency labelling schemes provide a unique representation for each member in the class. Specifically, an adjacency labelling scheme using $\Theta(\phi(n))$ bits in total is space-optimal for any family of size $2^{\Theta(\phi(n))}$. For any general function f, an f-labelling scheme does not necessarily provide a unique representation for each member in the class; for instance, let f be the boolean function that determines if two vertices are connected. That being said, there are functions which guarantee unique representations, for example, distance, so we are free to apply the concept of strong space-optimality to these labelling schemes.

2.2.2 Applications

The introduction of informative labelling schemes boosted the interest in the idea of space efficient distributed data structures as introduced by Muller [45] and Kannan et al. [31], given that different functions could be considered depending on the application. We describe three such applications here; a survey of informative labelling schemes can be found in [23], and further discussion of their applications can be found in [46], [32], [1], [4], and [56].

XML search engines

Informative labelling schemes have direct applications to the efficiency of XML (Extensible Markup Language) search engines [32]. A Web document conforming to the XML standard can be viewed as a tree with nested nodes corresponding to individual words, phrases, or sections of the document. Using informative labelling schemes, an XML search engine can assign labels to each of these nodes, thereby allowing relationships such as ancestor, parent, and sibling to be determined using only the labels of the nodes. This allows the search engine to answer web queries without repeatedly accessing the file itself. Moreover, by employing dynamic schemes the search engine will no longer have to recalculate the labels of the nodes when a small change is made to the XML document.

Routing Algorithms

Informative labelling schemes also have direct application to routing algorithms [51]. Consider sending a message along the best route from node v_0 to node v_f. Typically, v_0 consults a local routing table to determine the next node along the best path, say v_1, then sends the message to v_1, which repeats the process. Each node has a table consisting of n entries, where n is the number of nodes in the network. If we had an informative labelling scheme that could determine v_1 from the labels of v_0 and v_f, we could eliminate these routing tables, thereby reducing the amount of local storage required.

Broadcast Protocols

Informative labelling schemes can also be applied to broadcast protocols in much the same way that we applied them to routing algorithms [46]. Consider sending a message from node v_0 to node v_f, by means of network broadcast, as opposed to using an optimal route as discussed above. Typically, v_0 consults a local distance table to determine an upper bound on the distance to v_f so the broadcast can be terminated after a certain amount of time. As before, each node has a table consisting of n entries, where n is the number of nodes in the network. If we had an informative labelling scheme that could determine an upper bound on the distance between v_0 and v_f, then we could eliminate these distance tables, thereby reducing the amount of local storage required.

2.2.3 Previous work

Informative labelling schemes have been studied for a variety of functions besides adjacency; such functions include distance, nearest common ancestor and flow. As adjacency is the function of primary interest in this treatise, we will not present schemes for other functions; rather, we summarize the known results in Table 2.2. Specifically, Table 2.2 lists asymptotic bounds on the sizes of informative labelling schemes. We note that Table 2.2 does not contain information on schemes which approximate a function, nor schemes designed to encode multiple functions. To assist the reader, definitions of these classes can be found in Appendix A.

Table 2.2: Informative labeling schemes for functions other than adjacency († unless otherwise indicated, all sizes give number of bits per label; ‡ indicates a bound on the total number of bits for the entire labelling)

Function	**Family**	**Bound on scheme size**†
Ancestor	rooted trees	$O(\log n)$ [51]
Center	trees	$O(\log^2 n)$ [47]
		$\Omega(\log^2 n)$ [47]
Distance	binary trees	$\Omega(\log^2 n)$ [24]
	bipartite (smaller side of size k)	$\Omega(k(n-k) - O(n \log n))^{\ddagger}$[24]
	circular arc	$O(\log n)$ [22]
	cliquewidth-k	$O(\log^2 n)$ [13]
		$\Omega(\log^2 n)$ [24]
	cycles	$O(\log n)$ [47]
	maximum degree 3	$\Omega(n^{\frac{3}{2}})^{\ddagger}$ [24]
	degree-k	$\Omega(\sqrt{n})$ [24]
	degree-k planar	$\Omega(n^{\frac{4}{3}})^{\ddagger}$ [24]
	distance hereditary	$O(\log^2 n)$ [21]
		$\Omega(\log^2 n)$ [24]
	general graphs	$O(n)$ [24]
		$\Omega(n)$ [24]
	hypercubes	$O(\log n)$ [47]
	interval	$O(\log n)$ [22]
		$\Omega(\log n)$ [22]
	meshes	$O(\log n)$ [47]
	permutation	$O(\log^2 n)$ [34]
	planar	$O(\sqrt{n} \log n)$ [24]
		$\Omega(n^{\frac{1}{3}})$ [24]
	proper interval	$O(\log n)$ [22]
	recursive $r(n)$-separator	$O(r(n) \log n + \log^2 n)$ [24]
	tori	$O(\log n)$ [47]
	trees	$O(\log^2 n)$ [24]
		$\Omega(\log^2 n)$ [24]
	treewidth-k	$O(\log^2 n)$ [24]
		$\Omega(\log^2 n)$ [24]
	weighted binary trees (edge weights in $[0, M-1]$)	$\Omega(\log n \log M)$ [24]
	weighted c-decomposable (constant c; edge weights in $[0, M-1]$)	$O(\log^2 n + \log n \log M)$ [46]
	weighted k-outerplanar (constant k; edge weights in $[0, M-1]$)	$O(\log^2 n + \log n \log M)$ [46]
	weighted series-parallel (edge weights in $[0, M-1]$)	$O(\log^2 n + \log n \log M)$ [46]
	weighted trees (edge weights in $[0, M-1]$)	$O(\log^2 n + \log n \log M)$ [46]
	well (α, g)-separated	$O(g(n) \log n)$ [35]
Distance (at most d)	trees	$O(\log n + d\sqrt{\log n})$ [32]
Edge-connectivity	general graph	$O(\log^2 n)$ [33]

continued on next page ...

. . . continued from previous page

Function	**Family**	**Bound**
		$\Omega(\log^2 n)$ [33]
Flow	general graph (max edge capacity w)	$O(\log^2 n + \log n \cdot \log w)$ [33]
		$\Omega(\log^2 n + \log n \log w)$ [33]
Nearest Common Ancestor	rooted trees (return identifier)	$O(\log^2 n)$ [47]
		$\Omega(\log^2 n)$ [47]
	rooted trees (return label)	$O(\log n)$ [3]
Next Node Routing	forest	$O(n)$ [51]
	tree	$O(\log n)$ [56]
Reachability	planar digraph	$O(\log n)$ [55]
Separation Level	rooted trees	$O(\log^2 n)$ [47]
		$\Omega(\log^2 n)$ [47]
Steiner Tree	weighted graph, M bit edge weights	$O((M + \log n) \log n)$ [47]
		$\Omega((M + \log n) \log n)$ [47]
	weighted graph, arbitrary edge weights	$O((M + \log n) \log n)$ [47]
		$\Omega(M + n \log n)$ [47]
k-vertex connectivity (constant k)	general graph	$O(\log n)$ [33]
		$\Omega(\log n)$ [33]
k-vertex connectivity (k polylog in n)	general graph	$\Omega(k \log n)$ [33]

2.2.4 Dynamization

Although a significant number of results have appeared on the topic of informative labelling schemes, the seminal work of Kannan, Naor, and Rudich [31] made no mention of this variant of adjacency labelling schemes. Instead, Kannan et al. suggested the dynamic problem as a direction for future research. Unfortunately, the authors did not consider the problem in detail. At most, Kannan et al. suggest that the addition or deletion of a vertex or an edge should require only a "quick" update of the labels in order to obtain an adjacency labelling of the new graph.

In the following chapter, we develop the theory of dynamic informative labelling schemes. Specifically, we define what is meant by a dynamic informative labelling scheme, introduce the concept of error-detection, discuss the qualities that make a good dynamic scheme, and demonstrate the connection between error-detection and the graph recognition problem.

Chapter 3

Dynamic Schemes

In many applications the underlying topology is constantly changing and we desire algorithms which can accommodate these changes without having to process the new topology from scratch. By definition, informative labelling schemes are static; that is, the graph provided to the algorithm never changes. By studying the dynamic version of informative labelling schemes we hope to expand the applicability of informative labelling schemes to real world problems.

3.1 Dynamic adjacency labelling schemes

3.1.1 Definition

Before we define a dynamic informative labelling scheme, let us consider the following definition of a dynamic adjacency labelling scheme. This definition is based more on our intuitive understanding of the dynamization of a static problem; as such, it will be less precise than the formal definition we will encounter later.

Definition 3.1 *A dynamic adjacency labelling scheme of a family $\mathcal{G}$ of finite graphs is a tuple (M, D, Δ, R) for which*

- *(M, D) is an adjacency labelling scheme of $\mathcal{G}$.*
- *Δ is a set of functions which map graphs in $\mathcal{G}$ to other graphs.*
- *R is a polynomial time relabelling algorithm (relabeller) which, using only a vertex labelling, maintains an adjacency-correct labelling while a dynamic graph operation in Δ acts on a member of $\mathcal{G}$, providing the operation produces another graph in $\mathcal{G}$. Furthermore, R accesses vertex labels only as required.*

Moreover, we say that the dynamic adjacency labelling scheme is error-detecting if, given any input (δ, L_G), R is able to determine when $\delta(G) \notin \mathcal{G}$.

3.1.2 Example

In a dynamic informative labelling scheme the vertex labels must contain sufficient information to allow algorithms to update them. In general, the labels used in a static scheme do not contain enough information to be used in a dynamic scheme; however, the informative labelling schemes of certain classes are inherently dynamic.

Consider the adjacency labelling scheme described for trees in Section 2.1.1. For simplicity, let us require that the root not be deleted. In adding a new vertex, the relabeller chooses an identifier x, the smallest available natural number, then labels the new vertex $(x, parent(x))$. In deleting a vertex, the relabeller simply deletes its label from storage. Each relabelling can be performed in O(1) time (we also assume a word-level RAM computation model for the relabeller, where word sizes are $\Omega(\log n)$). Unfortunately, this dynamic scheme is not error-detecting because we cannot tell if the deletion of a vertex creates a disconnected forest; however, we can make the scheme error-detecting by adding, to each label, a counter of the number of children. Note that we can tell if the root is being deleted as $parent(v) = \text{NIL} \implies v = r$.

3.1.3 Underlying assumptions

Although the dynamic adjacency labelling scheme presented in Section 3.1.2 seems straightforward, there are two underlying problems.

1. It is possible to delete too many vertices, thereby causing the remaining labels to be too large (the point at which one decides that the labels are intolerably large depends on the application, as well as the family under consideration).

2. When a vertex is added and given an identifier, the relabeller must determine an acceptably short unused identifier to assign to it.

These problems do not depend on adjacency being the function under consideration; rather, they are inherent in any dynamic informative labelling scheme.

The obvious way to deal with the first problem is to relabel the graph from scratch using the marker algorithm. Note that we must initially use the decoder to determine a representation of the graph that the marker can use as input; for example, an adjacency matrix. This approach works well provided adjacency is the function under consideration. Using the decoder we can determine the adjacency of every pair of vertices, allowing us to reconstruct the graph and provide appropriate input to the marker. But what if the function under consideration is such that the decoder does not allow us to determine the graph? For example, what do we do with the boolean function that merely determines if two vertices are connected?

The obvious way to deal with the second problem is to maintain a centralized resource of identifiers that are not in use. Specifically, such a central resource might represent these identifiers using a list of intervals represented by their endpoints (in much the same manner as the interval representation presented in Chapter 1). But what if our application involves a distributed network? - does a new node have to broadcast to a central resource to get an identifier?

Given that we are approaching dynamic informative labelling schemes from a theoretical standpoint, we make certain assumptions to eliminate the problems discussed above. Specifically, we assume the following.

1. If n is the number of vertices presently in the graph, then there exists some constant k such that there has never been more than n^k vertices in the graph.
2. If an identifier is needed, a marker or relabeller can obtain the smallest available identifier in O(1) time.

Again, the validity of such assumptions is highly dependent on the application in which the dynamic scheme is being used. In our case, we do not want the restrictions of the application to hinder the development of the scheme. It is hoped that our dynamic schemes can be modified to work in different applications, with adjusted label sizes and running times as appropriate.

3.2 Dynamic informative labelling schemes

3.2.1 Definition

Let us now consider the formal definition of a dynamic f-labelling scheme.

Definition 3.2 *Consider a function $f(S, G)$ defined on sets of vertices S of fixed but arbitrary finite graphs G. A dynamic f-labelling scheme of a family $\mathcal{G}$ of finite graphs is a tuple (M, D, Δ, R) defined as follows.*

- *(M, D) is an f-labelling scheme of $\mathcal{G}$.*
- *Δ is a set of functions which map graphs in $\mathcal{G}$ to other graphs.*
- *R is a polynomial time relabelling algorithm (relabeller) whose input is a pair (δ, L_G), where $\delta \in \Delta$, $G \in \mathcal{G}$, and L_G is a (D, f)-correct labelling of V_G from $\mathcal{L}_G$ (defined shortly). Providing $\delta(G) \in \mathcal{G}$, R assigns a new (D, f)-correct labelling to $V_{\delta(G)}$ by accessing the labels of L_G only as required. Note that R need not be deterministic; accordingly, let $\mathcal{R}_{\delta, L_G}$ be the set of labellings of $V_{\delta(G)}$ which can be assigned by R on input (δ, L_G). For each G in $\mathcal{G}$ we define the family $\mathcal{L}_G$ of (D, f)-correct labellings of*

V_G by $L_G \in \mathcal{L}_G$ if and only if $L_G \in \mathcal{M}_G$ or there exists G^* in $\mathcal{G}$, δ in Δ, and L_{G^*} in $\mathcal{L}_{G^*}$ such that $\delta(G^*) = G$ and $L_G \in \mathcal{R}_{\delta, L_{G^*}}$.

Moreover, we say that the dynamic f-labelling scheme is error-detecting if, given any input (δ, L_G), R is able to determine when $\delta(G) \notin \mathcal{G}$.

In a less formal context, R can be considered as the composition of algorithms required by the graph operations found in Δ. For instance, if Δ permitted the addition or deletion of any edge from a graph, we might consider R to be comprised of two algorithms, InsertEdge(e,L_G) and DeleteEdge(e,L_G), which use a labelling L_G to relabel $G + e$ and $G - e$, respectively. Again, note that the algorithms InsertEdge and DeleteEdge do not directly receive G as input, rather, they are given access to the labels of the vertices of G as required. We are not prepared to maintain an adjacency matrix or adjacency lists to represent G; the goal of the informative labelling scheme is to efficiently represent G by doing away with such data structures. If adjacency is the function under consideration then maintaining an adjacency matrix or adjacency list would obviate the necessity of the adjacency labelling scheme! Moreover, in practice we are not interested in maintaining a labelling for every graph in the family; rather, we use the labelling of a graph to determine a labelling of a slightly modified graph, discarding the labelling of the original graph in the process. In this sense, we can omit the labelling from the input of the algorithms as these algorithms are directly modifying the labelling of the graph under consideration; that is, the above algorithms might be presented as InsertEdge(e) and DeleteEdge(e).

3.2.2 Assessing quality

Just as a static f-labelling scheme can be created for any function f when we allow sufficiently weak choices of M and D, sufficiently weak choices of M, D, Δ, and R will result in a dynamic scheme. Specifically, we can assess the quality of a dynamic scheme according to the following.

1. The quality of the static scheme (M, D) contained within.
2. The operations contained in Δ.
3. The running time of R.
4. The labels generated by R.
5. The labels modified by R.

Having previously discussed how to assess the quality of a static scheme in Section 2.1.2, we begin by considering the operations contained in Δ. Preferably, the labelling scheme will be fully dynamic; that is, Δ will contain the addition and deletion of a single edge or vertex (along with the edges incident with this vertex). In most cases these operations will allow us to transform any member of $\mathcal{G}$ into any other member of $\mathcal{G}$ without escaping the class $\mathcal{G}$. Specifically, if $\mathcal{G}$ is *hereditary*, that is, any vertex induced subgraph of a member is also a member, then these four dynamic operations are sufficient to transform any member of $\mathcal{G}$ into any other member of $\mathcal{G}$ without escaping $\mathcal{G}$. For each member G of $\mathcal{G}$, there is a sequence $S_G = \{G_0 = \emptyset, G_1, \ldots, G_{|V_G|-1}, G_{|V_G|} = G\}$ of members of $\mathcal{G}$ for which $G_{i-1} = G_i - v_i$, where v_i is a vertex of G_i and $1 \leq i \leq |V_G|$. Thereby, given $G^{(1)}, G^{(2)} \in \mathcal{G}$, we can construct $G^{(2)}$ from $G^{(1)}$ by using the vertex deletion relabeller to transform $G^{(1)}$ into $\emptyset$ via the members of $S_{G^{(1)}}$, then using the vertex addition relabeller to transform $\emptyset$ into $G^{(2)}$ via the members of $S_{G^{(2)}}$.

Along with the range of operations in Δ, we are also interested in the time taken by R on input (δ, L_G) relative to the time taken to label $\delta(G)$ by the fastest labelling algorithm of a static f-labelling scheme. Specifically, the purpose of the dynamic scheme is to provide quick updates of the labels; thereby, if there is a static scheme which can generate the labels in equal or better time, even from scratch, then there is no advantage gained by using the dynamic scheme.

As well, we might also judge a dynamic scheme according to the size of the labels generated by R. Naturally, the size of the labels generated by M is taken into account when judging the quality of the static scheme (M, D); however, the labels generated by M and R must be considered together, as labels from both algorithms could be in use at any given time. In particular, consider that the adjacency labelling scheme developed using adjacency matrices can be further developed into a dynamic adjacency labelling scheme. Since this dynamic scheme uses vertex labels of size $O(n)$, any other dynamic adjacency labelling scheme using labels of size $\Omega(n)$ would only be advantageous if it permitted faster updates of the labels than can be achieved using the dynamic scheme developed from adjacency matrices. Furthermore, we may wish to consider criteria such as balance and space-optimality, for appropriate functions f, just as we did for static schemes.

Aside from the size of the labels generated by R, we might also be interested in the labels that are changed by R. In some sense, this measure is captured in the running time of R; however, the running time does not give the full picture. In particular, we might like to know how many labels are changed and how the changes permeate through the graph. Perhaps, depending on the domain, these metrics could be more important than the label size. To measure this change, we define two quantities, *modification excess* and *modification locality*.

As we did with the definition of an informative labelling scheme, let us first consider these definitions intuitively in the context of dynamic adjacency labelling schemes. The modification excess of R is the maximum value, taken over all operations in Δ and all possible labellings produced in the dynamic adjacency labelling scheme, of the difference in cardinality between the set of vertices with modified labels and the set with modified neighbourhoods. In essence, the modification excess measures the number of vertices whose labels change even though we did not expect them to. The modification locality of R is the maximum value, taken over all operations in Δ and all possible labellings produced in the dynamic adjacency labelling scheme, of the maximum distance to the set of vertices with modified neighbourhoods from a vertex whose label has been modified, but whose neighbourhood has not. In essence, the modification locality measures the distance between the vertices whose labels we expected to change and the vertices whose labels we did not expect to change.

To illustrate how these measures are calculated, consider the relabelling depicted in Figure 3.1. Note that, when calculating these values for a dynamic scheme, we must consider the maximum of the values taken over *all* possible relabellings; here we consider a single isolated relabelling strictly for illustrative purposes. The neighbourhoods of b, c, and e change, whereas the labels of h, d, b, and e change. Therefore, the modification excess of this relabelling is $4 - 3 = 1$. Given that the labels of d and h are modified, but their neighbourhoods do not change, the modification locality of this relabelling is

$$\begin{aligned}\max_{x\in\{d,h\}}\{dist_G(x,\{b,c\}) &= \max_{x\in\{d,h\}} dist_{\delta(G)}(x,\{b,c,e\})\}\\ &= dist_G(h,\{b,c\})\\ &= dist_{\delta(G)}(h,\{b,c,e\})\}\\ &= 2.\end{aligned}$$

Note that, when adjacency is the function under consideration, the distances in G are the same as the distances in $\delta(G)$; however, this need not be the case in general. As such, we will take the minimum (of the maximum distances) over G and $\delta(G)$.

Integral to the definition of modification excess and modification locality for dynamic adjacency labelling schemes are the ideas of modified labels and modified neighbourhoods. We require analogous terms for arbitrary dynamic schemes, which we define below.

Definition 3.3 *Consider an input (δ, L_G) for the relabeller, R, of a dynamic f-labelling scheme (M, D, Δ, R) of a family $\mathcal{G}$, for which $\delta(G) \in \mathcal{G}$. Recall from Definition 3.2 that R*

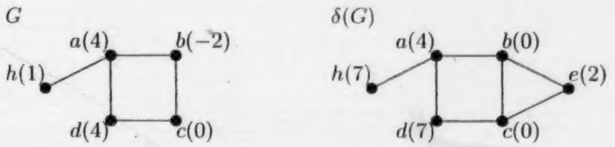


Figure 3.1: Adding a vertex to a graph (*labels in brackets*)

need not be deterministic; as such, let $\mathcal{R}_{\delta,L_G}$ *denote the set of all possible outputs* $L_{\delta(G)}$ *of* R *on input on* (δ, L_G).

We say that a vertex v *in* G *or* $\delta(G)$ *is* f*-changed if it belongs to some set of vertices,* S, *for which at least one of* $f(S,G)$ *and* $f(S,\delta(G))$ *is defined but* $f(S,G) \neq f(S,\delta(G))$; *in saying that* $f(S,G) \neq f(S,\delta(G))$ *we include the possibility that one of these expressions is undefined. We also say that the label of* v *is modified if at least one of* $L_{\{v\},G}$ *and* $L_{\{v\},\delta(G)}$ *is defined but* $L_{\{v\},G} \neq L_{\{v\},\delta(G)}$; *recall that* $L_{\{v\},G}$ *denotes the labelling of* v *in* G *and note that in saying* $L_{\{v\},G} \neq L_{\{v\},\delta(G)}$ *we also include the possibility that one of these expressions is undefined.*

Definition 3.4 *For each specific input/output pair* $((\delta, L_G), L_{\delta(G)})$, *we define its modification excess, denoted* $m_e((\delta, L_G), L_{\delta(G)})$, *to be the difference between the cardinality of the set of vertices whose labels are modified and the cardinality of the set of vertices that are* f*-changed. In turn, we define the modification excess of* (δ, L_G), *denoted* $m_e(\delta, L_G)$, *to be* $\max\limits_{L_{\delta(G)} \in \mathcal{R}_{\delta,L_G}} m_e((\delta, L_G), L_{\delta(G)})$ *and the modification excess of* R, *denoted* $m_e(R)$, *to be* $\max\limits_{(\delta,L_G)} m_e(\delta, L_G)$.

Definition 3.5 *For each specific input/output pair* $((\delta, L_G), L_{\delta(G)})$, *we define its modification locality, denoted* $m_l((\delta, L_G), L_{\delta(G)})$, *to be the minimum over* G *and* $\delta(G)$, *of the maximum over all the distances, from a vertex which is not* f*-changed, but whose label is modified, to the set of vertices which are* f*-changed; if there is no such vertex which is not* f*-changed then we let this value be zero. In turn, we define the modification locality of* (δ, L_G), *denoted* $m_l(\delta, L_G)$, *to be* $\max\limits_{L_{\delta(G)} \in \mathcal{R}_{\delta,L_G}} m_l((\delta, L_G), L_{\delta(G)})$ *and the modification locality of* R, *denoted* $m_l(R)$, *to be* $\max\limits_{(\delta,L_G)} m_l(\delta, L_G)$.

Having considered the relabeller to be the composition of several smaller relabelling algorithms, we can consider its modification locality and excess in terms of these smaller algorithms. Not only will this help us calculate these quantities, but this will also help us better understand the effect of specific dynamic changes on the labelling of the graph.

3.2.3 Graph recognition

In general, a graph recognition problem is of the form "Given a property P and a graph G, does G satisfy P?". More commonly, we consider questions of the form "Given a class $\mathcal{C}$ and a graph G, does G belong to $\mathcal{C}$?". Under the right circumstances, polynomial time recognition algorithms can be used to add error-detection to a dynamic scheme.

Theorem 3.6 *Consider a* $\Theta(B)$ *bit/label dynamic* f*-labelling scheme,* (M, D, Δ, R), *for a family of graphs* $\mathcal{G}$ *such that the following hold.*

- *For any graph G in $\mathcal{G}$, f allows us to uniquely determine G from any labelling in $\mathcal{L}_G$ (for example, consider adjacency or distance).*

- *The recognition problem is polynomial for $\mathcal{G}$; that is, there exists a polynomial algorithm A such that, for any graph G, A determines if G belongs to $\mathcal{G}$.*

Then $\mathcal{G}$ has an $\Theta(B)$ bit/label error-detecting dynamic f-labelling scheme.

Proof. On any input $(\delta, \mathcal{L}_G)$, R can use f to determine, in polynomial time, the structure of G and, hence, the structure of $\delta(G)$. In turn, R can incorporate A to determine if $\delta(G)$ is in $\mathcal{G}$, thereby making the dynamic scheme error-detecting. Moreover, there is no increase in the the number of bits used in a vertex label. ☐

In practice, the approach taken in the proof of Theorem 3.6 is an inefficient way of adding error detection to a dynamic scheme as we must incorporate a recognition algorithm into the relabeller. Rather, given a dynamic scheme, we can use our knowledge about recognition to inform us that an error-detecting dynamic scheme exists. Ideally, a slight modification of our dynamic scheme will give us error-detection. In Chapter 4, we develop an error-detecting dynamic adjacency labelling scheme for line graphs. Since the recognition problem had been shown polynomial for the class of line graphs [40, 50], we knew that any dynamic adjacency labelling scheme for line graphs could be made error-detecting. As discussed, our relabeller does not directly incorporate either of the polynomial recognition algorithms.

Of equal interest is the contrapositive of Theorem 3.6. Namely, if there is a dynamic f-labelling scheme for $\mathcal{G}$ such that

- for any graph G in $\mathcal{G}$, f allows us to determine the structure of G from any labelling in $\mathcal{L}_G$ (for example, adjacency or distance), and

- the relabeller cannot be augmented to make the scheme error-detecting,

then the recognition problem cannot be polynomial for $\mathcal{G}$.

Under the right circumstances, we can also develop recognition algorithms from error-detecting dynamic schemes.

Theorem 3.7 *Consider a family of graphs $\mathcal{G}$ for which there exists an error-detecting dynamic f-labelling scheme, (M, D, Δ, R). As well, let there be an algorithm which, for any graph G in $\mathcal{G}$, determines the following in polynomial time.*

- *A sequence $S_G = \{G_0 = G^*, G_1, \ldots, G_{k-1}, G_k = G\}$ of unique members of $\mathcal{G}$ for which $1 \leq k \leq |V_G|^c$, for some constant c.*

- *A sequence $G^{\Delta} = \{\delta_0, \delta_1, \ldots, \delta_{k-1}\}$ of members of Δ such that $\delta_i(G_i) = G_{i+1}$, for $0 \leq i \leq k-1$.*

- *A labelling of G^* which belongs to $\mathcal{L}_{G^*}$.*

Then graph recognition can be done in polynomial time for $\mathcal{G}$.

Proof. For any graph G in $\mathcal{G}$ we can transform the labelling of G^* into a labelling for G_{k-1} using a polynomial number of calls of the polynomial time algorithm R, namely $\{R_0, R_1, \ldots, R_{k-1}\}$, where $R_0 = R(\delta_0, L_{G^*})$ and $R_i = R(\delta_i, R_{i-1})$, for $1 \leq i \leq k-1$. We can now resolve the membership of G in $\mathcal{G}$ according to the action of R when it attempts to determine a labelling of G from the labelling of G_{k-1}. If $G \in \mathcal{G}$, then R will determine a labelling of G; otherwise, it will output that $G \notin \mathcal{G}$ since it is an error-detecting algorithm. $\square$

An interesting corollary of Theorem 3.7 follows when we consider hereditary classes. Recall that a class of graphs is said to be *hereditary* if every vertex induced subgraph of every member is also a member.

Corollary 3.8 *Consider a hereditary family of graphs $\mathcal{G}$ for which there exists an error-detecting dynamic f-labelling scheme, (M, D, Δ, R), where Δ includes the addition of vertices (along with incident edges), and R adds vertices in $O(X)$ time. Graph recognition can be done in $O(nX)$ time for $\mathcal{G}$.*

Proof. For any graph G, consider a sequence $S_G = \{G_0 = \emptyset, G_1, \ldots, G_{|V_G|-1}, G_{|V_G|} = G\}$ of graphs (the graphs may be disconnected) for which $G_{i-1} = G_i - v_i$, where v_i is some vertex of G_i and $1 \leq i \leq |V_G|$. As well, consider the corresponding sequence $S'_G = \{X_0 = \emptyset, \ldots, X_{|V_G|-1}\}$, where X_i is the set of vertices in G_i to which v_{i+1} is adjacent. These sequences can be determined in $O(n)$ time.

Since $\mathcal{G}$ is hereditary, G is in $\mathcal{G}$ if and only if every G_i is in $\mathcal{G}$. Starting with the empty labelling for $\emptyset$, R can determine a labelling for G_{i+1} from that of G_i, in $O(X(G))$ time. Because our scheme is error-detecting, R will detect if some G_i does not belong to $\mathcal{G}$. Consequently, we have an $O(nX)$ time recognition algorithm for $\mathcal{G}$. $\square$

One might be led to believe that we have just argued that the recognition problem is polynomial for any hereditary class of graphs, but this is not the case. It is true that all hereditary families have a dynamic adjacency labelling scheme, namely, the default scheme obtained from adjacency matrices. However, there is no guarantee that such a scheme is error-detecting. The fact that the dynamic scheme is error-detecting is critical in the proof of Corollary 3.8.

We use Corollary 3.8 in Chapter 5 to establish $O(r^3n^3)$ time recognition for r-minoes and r-bics. Although polynomial time recognition has already been established for r-minoes by Johnson, Yannakakis, and Papadimitriou [30], as well as Metelsky and Tyshkevich [44], our approach is faster. A slight modification of the algorithm of Johnson et al. which generates all maximal cliques in lexicographic order, results in an $O(rn^4)$ recognition algorithm. Metelsky and Tyshkevich do not explicitly state the asymptotic time they require to perform recognition, but their forbidden subgraph approach requires $\Theta(n^{r+2})$ time.

3.2.4 Previous work

The dynamic version of adjacency labelling schemes was mentioned by Kannan et al. [31], but they did not consider the problem in detail. The first paper to address this dynamic problem is that of Brodal and Fagerberg [10] who develop a dynamic adjacency labelling scheme for graphs of bounded arboricity. Their relabelling algorithm keeps an outneighbourhood list for each vertex v, similar to that seen in Section 2.1.1, and it also includes a mechanism to handle outdegree lists which get too big. On a graph with n vertices and arboricity bounded by c, Brodal and Fagerberg's representation supports adjacency testing in $O(c)$ time, edge insertions in $O(1)$ time, and edge deletions in $O(c+\log n)$ time. We present their algorithms for handling the addition and deletion of a single edge from a graph of bounded arboricity c in Figure 3.2 and note that these algorithms are easily modified to handle the addition and deletion of vertices. Unfortunately, these algorithms are built on the assumption that the changes to the graph do not cause its arboricity to exceed c; that is, they are not error-detecting. In their article, Brodal and Fagerberg do describe modified algorithms which are error-detecting and can handle unspecified arboricities, but these algorithms have higher asymptotic running times.

```
INSERT($u, v$)
    $u.adj[] \leftarrow u.adj[] \cup \{v\}$
    if $|u.adj[]| = 4c + 1$ then
        $S \leftarrow \{u\}$
        while $S \neq \emptyset$ do
            $w \leftarrow$ POP($S$)
            for $x \in w.adj[]$ do
                $x.adj[] \leftarrow x.adj[] \cup \{w\}$
                if $|x.adj[]| = 4c + 1$ then
                    PUSH($S, x$)
            $w.adj[] \leftarrow \emptyset$

DELETE($u, v$)
    $u.adj[] \leftarrow u.adj[] \setminus \{v\}$
    $v.adj[] \leftarrow v.adj[] \setminus \{u\}$
```

Figure 3.2: Algorithms for dynamic adjacency labelling of graphs of bounded arboricity c

Since the work of Brodal and Fagerberg[10], there have been few other works to examine dynamic informative labelling schemes. Cohen, Kaplan, and Milo [11] consider the ancestor function on rooted trees in which new vertices can be added to the tree. Contrary to the problem which we are considering, Cohen et al. require that the labels assigned to vertices be persistent, that is, the label cannot be changed once it has been assigned. As such, their vertex labels may be larger than those required using our model, as we have the freedom to modify inefficient labels. Specifically, the scheme of Cohen et al. uses $O(n)$ bit labels for arbitrary trees, and $O(d \log \Delta)$ bit labels for trees with maximum depth d and maximum degree Δ. In each case they prove these labellings to be optimal, given the persistency requirement. Although the authors do not explicitly discuss the running time of their relabeller (which is dramatically simpler, given that vertices cannot be relabelled), it is $O(1)$ in each case. Additionally, Cohen et al. consider the scenario in which the (re)labeller is given clues about the future structure of the tree, which lead to smaller labels.

Two interesting results on dynamic distance labelling schemes can be found in a paper by Korman, Peleg, and Rodeh [39] (an earlier version appears as [38]). First, the authors devise a dynamic distance labelling for unweighted trees, allowing the addition and deletion of leaves, that uses $O(\log^2 n)$ bit labels, which is optimal even for the static problem [24]. Like Cohen et al. [11], Korman et al. do not explicitly state the running time of their relabeller (which is actually part of a more complicated marker). However, they do discuss the notions of message complexity and bit complexity, which are the maximum number of messages and bits, respectively, that must be sent when the graph changes. Specifically, their scheme has a $O(\log^2 n)$ amortized message complexity and $O(\log^2 n \log\log n)$ amortized bit complexity, thereby, even a relabeller sending sequential messages would run in $O(\log^2 n \log\log n)$ time.

Second, they establish a framework for extending static schemes to dynamic schemes; this framework can be used for a variety of functions including distance, separation level, and flow. For the partially dynamic scenario in which vertices can only be added, this framework causes a $O(\log n)$ multiplicative increase in the label size and the amortized message complexity (as an increase over the running time of the static marker). If an upper bound on n is known in advance, then the multiplicative factors on label size and amortized message complexity reduce to $O(\log^2 n/ \log\log n)$ and $O(\log n/ \log\log n)$, respectively. For the fully dynamic model in which vertices can be added and deleted, there is also an additive increase in the amortized message complexity (in addition to those multiplicative increases previously mentioned). The authors do not offer any comment on the change in amortized bit complexity for either of these scenarios.

As a follow up to Korman et al. [39], Korman and Peleg [37] consider dynamic schemes that approximate distances in weighted trees and cycles. Dynamism is achieved by allowing the weights of the edges to vary, where w is the maximum edge weight. For the increasing dynamic scenario, in which edge weights can only increase, their schemes use $O(\log^2 n + \log n \log w)$ bit labels, which is optimal even for exact distances in the static setting [24]. Moreover, the message and bit complexities of the relabeller are $O(m \log^2 n + n \log n \log m)$

and $O(m \log^2 n \log\log n + n \log n \log m \log\log n)$, respectively, where m is the number of edges whose weights change. For the fully dynamic scenario, in which edge weights can both increase and decrease, labels require $O(\log^2 n + \log n \log w)$ bits, and the message and bit complexities are $O(m\Lambda \log^2 n)$ and $O(m\Lambda \log^2 n \log\log n)$, respectively, where $\Lambda = \max\{\frac{B(e,d)}{d} : d \geq 1, e \in E_G\}$ and $B(e,d)$ is the number of vertices at distance at most d from and endpoint of e. In the fully dynamic scenario, if the graph is a path or cycle, then the label size reduces to $O(\log n \log m)$ bits and the message complexity reduces to $O(m \log^2 n)$.

3.3 New dynamic schemes

Having presented the theory of dynamic informative labelling schemes, the remainder of the thesis focuses on the development of new dynamic adjacency labelling schemes for a selection of classes. Our presentation of each of these dynamic schemes carefully describes the labelling/marker, the decoder, and the relabeller. The relabeller is presented in parts, one for each of the allowable actions: vertex deletion, vertex addition, edge deletion, and edge addition. Moreover, in the body of the thesis we describe the relabeller at a high level, saving the detailed pseudocode for Appendix C.

Specifically, in Chapter 4, we present a dynamic adjacency labelling scheme for line graphs. Our dynamic scheme for line graphs uses $O(\log n)$ bit labels and updates in $O(e)$ time, where e is the number of edges added to, or deleted from, the line graph.

In Chapter 5, we develop a $O(r \log n)$ bit/label dynamic adjacency labelling scheme for r-*minoes*, graphs with no vertex in more than r maximal cliques. Edge addition and deletion can be handled in $O(r^2\mathbf{D})$ time, vertex addition in $O(r^2e^2)$ time, and vertex deletion in $O(r^2e)$ time, where $\mathbf{D}$ is the maximum degree of the vertices in the original graph and e is the number of edges added to, or deleted from, the original graph. As well, we develop a $O(r \log n)$ bit/label dynamic adjacency labelling scheme for r-*bics*, graphs with no vertex in more than r maximal bicliques. Edge addition and deletion, as well as vertex deletion, can be handled in $O(r^2\mathbf{B})$ time, and vertex addition in $O(r^2 n\mathbf{B})$ time, where $\mathbf{B}$ is the size of the largest biclique in the original graph.

Finally, in Chapter 6, we present a dynamic adjacency labelling scheme for proper interval graphs, which are a subclass of interval graphs. Our dynamic scheme for proper interval graphs uses $O(\log n)$ bit labels and handles all operations in $O(n)$ time.

Chapter 4

Line graphs

Recall from Section 2.1.1, the definition of a line graph and its base.

Definition 4.1 *Given a graph $G = (V_G, E_G)$, its line graph is the graph $L(G) = (E_G, E_{L(G)})$ for which $\{u, v\} \in E_{L(G)}$ if and only if u and v are adjacent edges in G.*

In this chapter, we present a dynamic adjacency labelling scheme for line graphs that allows the addition and deletion of vertices and edges. The labels used in this scheme require $O(\log n)$ bits, and updates require $O(e)$ time, where e is the number of edges added to or deleted from the line graph. In comparison, the best known (static) adjacency labelling scheme for line graphs, presented in Section 2.1.1, uses $O(\log n)$ bit labels and requires $\Theta(n)$ time to generate a labelling [45].

Given the simplicity of their intersection representation, line graphs are perhaps the most fundamental intersection class. As such, we hope that our dynamic adjacency labelling scheme for line graphs will offer insight into the development of dynamic schemes for other intersection classes.

4.1 Partition isomorphism

As mentioned in Section 2.1.1, Whitney [58] has shown that every connected line graph has a unique base, up to isomorphism, except for K_3 which has two bases, namely, K_3 and $K_{1,3}$. Just as a graph "generates" a line graph, we can can say that an edge labelled graph "generates" a vertex labelled line graph. For this reason, we will also use the term "base" to refer to an edge labelled graph, with no isolated vertices, that generates a particular vertex labelled line graph.

Our work on line graphs requires a concept similar to isomorphism, but involving edge labellings. Given an edge labelling ψ of a graph (in which each label is unique), for each edge label α, we let P_α^ψ denote the partition of the labels adjacent to α that is determined by the endpoints of α. We define two bases of a vertex labelled line graph $L(G)$, having edge labellings ψ_1 and ψ_2, to be *partition isomorphic* if the bases are isomorphic and $P_\alpha^{\psi_1} = P_\alpha^{\psi_2}$, for all edge labels α. For example, the two bases shown in Figure 4.1(d) are not partition isomorphic; in one of these bases, the partition corresponding to a is $\{\{b\}, \{c\}\}$, while in the other it is $\{\{b, c\}, \emptyset\}$.

When we consider the theorem of Whitney in the context of labelled line graphs, we arrive at the following theorem.

Theorem 4.2 *Every connected vertex labelled line graph, except those shown in Figure 4.1(a), has a unique (edge labelled) base, up to partition isomorphism. For each of the four exceptions, a vertex labelled graph has two bases that are not partition isomorphic.*

Proof. Consider a connected vertex labelled line graph $L(G)$ which has two (edge labelled) bases, G_1 and G_2, that are not partition isomorphic. Fundamental to this proof is the fact that the edge adjacencies in G_1 and G_2 are identical, that is, two edges are adjacent in G_1 if and only if they are adjacent in G_2.

Let the labellings of G_1 and G_2 be ψ_1 and ψ_2, respectively, and let a be a label for which $P_a^{\psi_1} \neq P_a^{\psi_2}$. Moreover, let $P_a^{\psi_1} = \{Q_{\psi_1}, R_{\psi_1}\}$ and $P_a^{\psi_2} = \{Q_{\psi_2}, R_{\psi_2}\}$. Trivially, observe that $|Q_{\psi_1}| + |R_{\psi_1}| \geq 2$, otherwise, $P_a^{\psi_1} = P_a^{\psi_2}$.

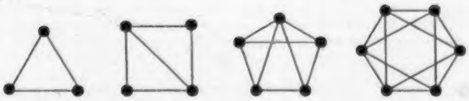
Now consider when one of $|Q_{\psi_1}|$, $|R_{\psi_1}|$, $|Q_{\psi_2}|$, or $|R_{\psi_2}|$ is at least three; without loss of generality, let $b, c, d \in Q_{\psi_1}$. We first consider the case when $\{b, c, d\} \subseteq Q_{\psi_2}$ or $\{b, c, d\} \subseteq R_{\psi_2}$; without loss of generality, assume the former. Since $P_a^{\psi_1} \neq P_a^{\psi_2}$, there must be a label e that belongs to Q_{ψ_1}, but not to Q_{ψ_2}, or vice versa; again, without loss of generality, assume the former. Given that $e \in Q_{\psi_1}$, e is adjacent to each of b, c, and d in both G_1 and G_2. Yet $e \in R_{\psi_2}$, so G_2 must contain each of the three cycles of edges abe, ace, and ade, which is not possible unless $b = c = d$. Next, consider the case when $\{b, c, d\} \not\subseteq Q_{\psi_2}$ and $\{b, c, d\} \not\subseteq R_{\psi_2}$; without loss of generality, assume that $\{b, c\} \subseteq Q_{\psi_2}$ and $\{d\} \subseteq R_{\psi_2}$. A similar argument gives that G_2 must contain both of the three cycles of edges abd and acd, which is not possible unless $b = c$.

Having shown that neither Q_{ψ_1}, R_{ψ_1}, Q_{ψ_2}, nor R_{ψ_2} can contain three edges, we observe that the only way that $P_a^{\psi_1} \neq P_a^{\psi_2}$ is if, without loss of generality, there exist edges b and c such that $\{b, c\} = Q_{\psi_1}$, $b \in Q_{\psi_2}$, and $c \in R_{\psi_2}$. Since b and c are adjacent in G_1, they are also adjacent in G_2; as such, the edges a, b, and c form a $K_{1,3}$ in G_1 and a K_3 in G_2.

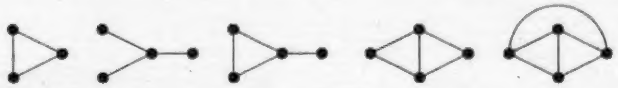
At this point, we resolve our proof into three cases.

- $Q_{\psi_1} = \{b, c\}$, $R_{\psi_1} = \emptyset$, $Q_{\psi_2} = \{b\}$, and $R_{\psi_2} = \{c\}$. If $V_{L(G)} = \{a, b, c\}$, then G_1 and G_2 are as shown in Figure 4.1(c), and $L(G) = K_3$, as desired. Moreover, since the set $\{b, c\}$ exhibits only two distinct partitions, G_1 and G_2 are the only bases of $L(G)$.

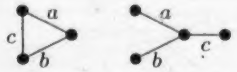
  If $V_{L(G)} \supset \{a, b, c\}$, then, without loss of generality, there must be some label d adjacent to both b and c, but not a, as the edges a, b, and c form a K_3 in G_2. Since the edges



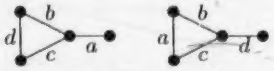
(a) The only connected line graphs with two (edge labelled) bases that are not partition isomorphic.



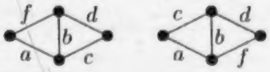
(b) The base graphs of the four line graphs pictured in Figure 4.1(a).



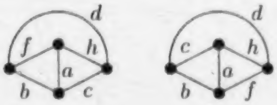
(c) Two edge labelled bases of the line graph K_3 that are not partition isomorphic.



(d) Two edge labelled bases of the line graph $K_4 - e$ that are not partition isomorphic.



(e) Two edge labelled bases of the line graph $K_5 - m$, where m is a maximal matching, that are not partition isomorphic.



(f) Two edge labelled bases of the line graph $K_6 - m$, where m is a maximal matching, that are not partition isomorphic.

Figure 4.1: Partition isomorphism of graphs

a, b, and c form a $K_{1,3}$ in G_1, the edges b, c, and d form a K_3 in G_1. Therefore, d is the unique label adjacent to both b and c, but not a, in G_2.

Now, if $V_{L(G)} \supset \{a, b, c, d\}$, then, without loss of generality, there must be some label z adjacent to b and d, but neither a nor c, as G_1 is the graph shown in the left hand side of Figure 4.1(d). However, given the configuration of G_2, as shown in the right hand side of Figure 4.1(d), these adjacencies are impossible.

With $V_{L(G)} = \{a, b, c, d\}$, G_1 and G_2 are as shown in Figure 4.1(d), and $L(G) = K_4 - e$, as desired. Moreover, the partitions $P_a^{\psi_1}$ and $P_a^{\psi_2}$ fix $P_b^{\psi_1}$, $P_b^{\psi_2}$, $P_c^{\psi_1}$, $P_c^{\psi_2}$, $P_d^{\psi_1}$, and $P_d^{\psi_2}$. Therefore, since the set $\{b, c\}$ exhibits only two distinct partitions, G_1 and G_2 are the only bases of $L(G)$.

- $Q_{\psi_1} = \{b, c\}$, $R_{\psi_1} = \{f\}$, $Q_{\psi_2} = \{b, f\}$, and $R_{\psi_2} = \{c\}$. Just as a, b, and c form a K_3 in G_2, a, b, and f form a K_3 in G_1. Therefore, if $V_{L(G)} = \{a, b, c, f\}$, then $L(G) = K_4 - e$, as desired. Again, $P_a^{\psi_1}$ and $P_a^{\psi_2}$ fix the remaining partitions, so G_1 and G_2 are the only bases of $L(G)$.

  Again, if $L(G)$ contains an additional vertex, then it can only be the vertex d discussed above. In this case, G_1 and G_2 are as shown in Figure 4.1(e), and $L(G) = K_5 - m$, as desired, where m is a maximal matching. Moreover, $P_a^{\psi_1}$ and $P_a^{\psi_2}$ fix the remaining partitions. Therefore, since the set $\{b, c\}$ exhibits only two distinct partitions, G_1 and G_2 are the only bases of $L(G)$.

- $Q_{\psi_1} = \{b, c\}$, $R_{\psi_1} = \{f, h\}$, $Q_{\psi_2} = \{b, f\}$, and $R_{\psi_2} = \{c, h\}$. Just as a, b, and c form a K_3 in G_2 and a, b, and f form a K_3 in G_1, a, c, and h form a K_3 in G_1 and a, f, and h form a K_3 in G_2. Therefore, if $V_{L(G)} = \{a, b, c, f, h\}$, then $L(G) = K_5 - m$, as desired, where m is a maximal matching. Again, $P_a^{\psi_1}$ and $P_a^{\psi_2}$ fix the remaining partitions, therefore, G_1 and G_2 are the only bases of $L(G)$.

  Again, if $L(G)$ contains an additional vertex, then it can only be the vertex d discussed above. In this case, G_1 and G_2 are as shown in Figure 4.1(f), and $L(G) = K_6 - p$, as desired, where m is a maximal matching. Moreover, the partitions $P_a^{\psi_1}$ and $P_a^{\psi_2}$ fix the remaining partitions. Therefore, since the set $\{b, c\}$ exhibits only two distinct partitions, G_1 and G_2 are the only bases of $L(G)$.

$\square$

4.2 The dynamic scheme

4.2.1 Vertex labels, marker, and decoder

Our dynamic adjacency labelling scheme for line graphs builds upon the adjacency labelling scheme for line graphs found in Section 2.1.1. Given a line graph $L(G)$, each vertex of the line graph is assigned a unique identifier from $\{1, \ldots, |V_{L(G)}|\}$. These vertex identifiers give rise to an edge labelling of some base G, from which we will derive the remainder of our labelling. Like the adjacency labelling scheme of Muller [45], we also assign each vertex of G a unique identifier from $\{1, \ldots, |V_G|\}$. For simplicity, we refer to vertices by their identifiers.

Our dynamic scheme uses graph substructures and circular doubly linked lists to distribute information about the neighbourhood of a vertex in the line graph over the labels of the neighbours. Specifically, for each vertex in the base, we maintain a circular doubly linked list of the edges incident with that vertex. For each edge v in G, the circular doubly linked lists associated with its endpoints partition the edges adjacent to v, exactly as seen in our discussion of partition isomorphism, with the singular exception of v itself. Moreover, the union of the two circular doubly linked lists associated with the endpoints of v, give the vertices adjacent to v (in $L(G)$).

Given a vertex v, its label will consist of the following information (in addition to its identifier).

$v.ep_0, v.ep_1$: Considered as an edge in the base, v has two endpoints; $v.ep_0$ and $v.ep_1$ are the identifiers of these endpoints.

$v.nn_0, v.nn_1$: The values of $|N(v.ep_0)|$ and $|N(v.ep_1)|$ (in the base), respectively, where $N(x)$ denotes the open neighbourhood of the vertex x.

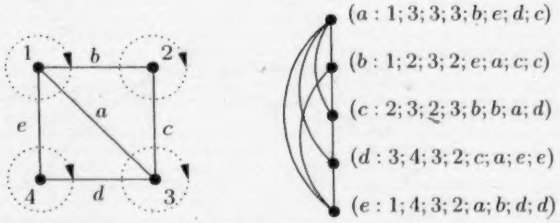
$v.prev_0, v.prev_1, v.nx_0, v.nx_1$: With v as the current edge in the circular doubly linked list about $v.ep_i$, the identifiers of the previous and next edges are $v.prev_i$ and $v.nx_i$, respectively.

In particular, the label of a vertex is (v: $v.ep_0$; $v.ep_1$; $v.nn_0$; $v.nn_1$; $v.prev_0$; $v.nx_0$; $v.prev_1$; $v.nx_1$), as illustrated in Figure 4.2. Like the static scheme of Muller, we assume that the marker knows the structure of G, so that it can generate an initial labelling in $\Theta(n+m)$ time, using a breadth first search. Otherwise, the marker must use an algorithm like that of Lehot [40] or Roussopoulos [50] to determine G from $L(G)$. Also like the static scheme of Muller, the decoder can determine the adjacency of v_1 and v_2 in $O(1)$ time, using only their labels, by checking if $\{v_1.ep_0, v_1.ep_1\} \cap \{v_2.ep_0, v_2.ep_1\} = \emptyset$.

Consider a line graph with n vertices. If $\overline{string}$ denotes the number of bits required to represent *string* then the number of bits used in the label of v is

$$\overline{v} + \sum_{i=0}^{1} \left(\overline{v.ep_i} + \overline{v.nn_i} + \overline{v.prev_i} + \overline{v.nx_i} \right).$$

In Section 3.1.3, we observed that if the deletion of vertices is permitted, it may result in the identifiers of the remaining vertices not being space-optimal. The same is true for any


1
b
2
a
e
c
4
d
3
(a : 1; 3; 3; 3; b; e; d; c)
(b : 1; 2; 3; 2; e; a; c; c)
(c : 2; 3; 2; 3; b; b; a; d)
(d : 3; 4; 3; 2; c; a; e; e)
(e : 1; 4; 3; 2; a; b; d; d)

(a) G (↻ *denotes the order of the circular linked list*).

(b) Labels of $L(G)$.

Figure 4.2: Our dynamic adjacency labelling scheme for line graphs

labelling in which identifiers are assigned. As such, let the largest identifier of a vertex in the line graph be L_1, and let the largest identifier of a vertex in the base graph be L_2. Thereby, $l(v), l(v.prev_i), l(v.nx_i) \in O(\log L_1)$ and $l(v.ep_i) \in O(\log L_2)$. Moreover, since the base has no isolated vertices, $|V_G| \leq 2|E_G| \leq 2|V_{L(G)}| \leq 2n$; therefore, $l(v.nn_i) \in O(\log n)$, and the label of v uses $O(\log L_1 + \log L_2 + \log n)$ bits. If L_1 and L_2 are polynomial in n, which we stated as an assumption in Section 3.1.3, then the label size of v reduces to $O(\log n)$. In turn, the graph is represented using $O(n \log n)$ bits.

Using an argument found in a recent text of Spinrad [53] (p. 18), we can show that there are $2^{\Omega(n \log n)}$ labelled line graphs on n vertices. Thereby, the dynamic scheme is strongly space-optimal for labelled line graphs. Consider a graph with $\frac{n}{2}$ disjoint edges, each of which has one endpoint in $\{1, \ldots, \frac{n}{2}\}$ and the other in $\{\frac{n}{2} + 1, \ldots, n\}$. There are $(\frac{n}{2})!$ such graphs, each of which is a line graph, yet

$$(\frac{n}{2})! > \frac{(\frac{n}{2})!}{(\frac{n}{4})!} > (\frac{n}{4})^{\frac{n}{4}} = 2^{\frac{n}{4}\log(\frac{n}{4})} \in 2^{\Omega(n \log n)}.$$

This scheme may also be space-optimal for unlabelled line graphs; however, this lower bound has not yet been established in the unlabelled case.

The success of our dynamic scheme lies in the ability to change the labelling of a graph to reflect another partition non-isomorphic base, when necessary. In particular, if a line graph has a connected component with two bases that are not partition isomorphic, then it is possible that the labelling derived from one of these bases will permit certain dynamic operations while the other will not. For instance, consider the two bases depicted in Figure 4.1(d). If we wish to add a new vertex v to the corresponding line graph such that its neighbours are a and b, then the equivalent operation in the base is the addition of an edge that is adjacent to only the edges labelled a and b. This can be done using one of the bases, but not the other. Again, we note that in any informative labelling scheme we have access to the vertex labels only. Consequently, when we say that we change the base, we ultimately mean that we change the labelling of the graph so as to reflect a new base.

Even more critical to the success of our dynamic scheme is the inclusion of sufficient information in the labels to deduce, at least partially, the structure of the base. Upon modification of the line graph, our knowledge of the original base will allow us to determine the base of the new line graph and, hence, the labels of the new line graph. To illustrate this need for knowledge about the base, consider the line graphs presented in Figure 4.3. Even though the line graph $\delta(L(G))$ is formed by deleting a single edge from $L(G)$, the change in the base, from G to G', is substantial; in particular, the required change affects much more than just the edges of the base that correspond to the endpoints of the deleted edge in the line graph. If our dynamic scheme were to use the labels of the static scheme of Muller [45], then it would be impossible to deduce the neighbourhood of a vertex v without checking the label of every vertex u to see if the edges of the base corresponding to u and v share a common endpoint.

Of particular interest is how we can use the vertex labels to traverse circular doubly linked lists. For any edge v of G, we know that the next vertex in the circular doubly linked list $\mathcal{L}$ about $v.ep_i$ is $v.nx_i$. Let $u = v.nx_i$. Ideally, the next vertex in $\mathcal{L}$ after u would be $u.nx_i$, however, it could be either $u.nx_0$ or $u.nx_1$. Consequently, before we proceed, we must determine which of $u.ep_0$ and $u.ep_1$ is $v.ep_i$; fortunately, this simple test requires O(1) time. As such, $\mathcal{L}$ can be traversed in $O(v.nn_i)$ time. For simplicity, we will say that $\mathcal{L}$ can be traversed in $O(|\mathcal{L}|)$ time, where $|\mathcal{L}|$ is the number of edges in $\mathcal{L}$.

4.2.2 Relabeller

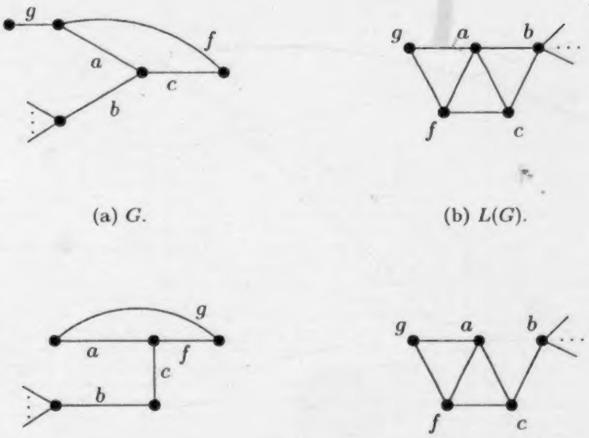
In the remainder of this chapter, we present the relabeller that belongs to our dynamic adjacency labelling scheme for line graphs. For each graph operation, a "gentler" version of the relabelling algorithm will be discussed in this chapter, with detailed pseudocode appearing in Appendix C.

For simplicity, we will refer to $\delta(L(G))$ as $L(G')$, the line graph with base G', yet we implore the reader to recognize that it is the graph $L(G)$ to which the operation δ is being applied. We are not applying δ to G to get G', rather, G' is the resulting base graph of $L(G') = \delta(L(G))$.

Deleting a vertex

As with all of our graph modifications, it is imperative to understand how a change in the line graph causes a change in the base. By deleting a vertex from the line graph, we delete the corresponding edge in the base, as depicted in Figure 4.4.

Let v be the vertex to be deleted, and let X be the neighbours of v in $L(G)$. For each endpoint $v.ep_i$ of v (in G), DELETEVERTEX, the algorithm presented in Figure 4.5, first



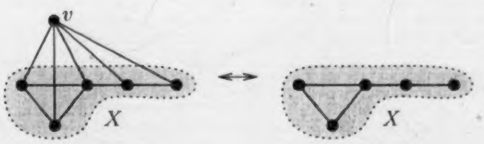
(a) G.

(b) $L(G)$.

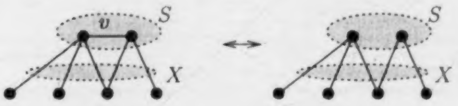
(c) G'.

(d) $L(G') = \delta(L(G))$.

Figure 4.3: An edge is deleted from (or added to) a line graph. The use of ellipses indicates that the graph extends arbitrarily from the indicated vertex



(a) A vertex is deleted from (or added to) $L(G)$



(b) The corresponding change in G. The set S is a valid set

Figure 4.4: A vertex is deleted from (or added to) the line graph

determines whether v is the only edge incident with $v.ep_i$. It does this in O(1) time by testing the condition $v.nn_i = 1$. If v is the only edge incident with $v.ep_i$, then DELETEVERTEX frees the identifier of $v.ep_i$, which takes O(1) time.

If there are other edges incident with $v.ep_i$, then DELETEVERTEX traverses $\mathcal{L}_i$, the circular doubly linked lists about $v.ep_i$, decrementing the $.nn$ counter that corresponds to $\mathcal{L}_i$ by one, for every vertex l (in $L(G)$) in $\mathcal{L}_i$. Traversing $\mathcal{L}_i$ to decrement these counters takes $\Theta(|\mathcal{L}_i|)$ time. Once these counters have been adjusted, v is removed from $\mathcal{L}_i$, which takes O(1) time.

Once both endpoints of v have been addressed, v is deleted and its identifier is freed for future use. This elimination takes O(1) time. Since, $|\mathcal{L}_1| + |\mathcal{L}_2| = |X| + 2$, DELETEVERTEX runs in $\Theta(|X|) \in O(n)$ time. Moreover, DELETEVERTEX accesses $\Theta(|X|)$ vertex labels, requiring a total of $\Omega(|X|)$ bits; therefore, the running time of DELETEVERTEX is polynomial in the size of its inputs. Moreover, DELETEVERTEX is error-detecting because the class of line graphs is hereditary.

DELETEVERTEX$(L(G), v)$

Input: An adjacency labelling of a line graph $L(G)$ (that is, the labels thereof) created using our dynamic scheme, and a vertex v in $V_{L(G)}$. Note that the labels of $L(G)$ are only accessed as required.

Output: An adjacency labeling of a graph $L(G')$ (again, the labels thereof) formed by deleting v from $L(G)$, providing $L(G')$ is a line graph. If $L(G')$ is not a line graph, then the output indicates as such.

```
for $i \leftarrow 0$ to 1 do
    if $v$ is the only edge incident with $v.ep_i$ then
        free the identifier of $v.ep_i$
    else $\mathcal{L}_i \leftarrow$ the circular doubly linked list about $v.ep_i$
        for $l \in \mathcal{L}_i$ do
            decrement $l$'s counter of the number of edges in $\mathcal{L}_i$ by 1
        remove $v$ from $\mathcal{L}_i$
delete $v$ and free its identifier
```

Figure 4.5: The relabeller DELETEVERTEX which relabels the line graph when a vertex is deleted

Proposition 4.3 *The modification excess and modification locality of* DELETEVERTEX *are zero.*

Proof. First, observe that the set of vertices whose neighbourhoods change is $X \cup \{v\}$. If the label of a vertex x is modified, then its corresponding edge in the base had been in one of the circular linked lists about an endpoint of v (in G). That is, $x \in X \cup \{v\}$. Therefore, the set of vertices with modified labels is a subset of the set of vertices whose neighbourhoods change, giving the desired result. □

Adding a vertex

Adding a vertex to the line graph, along with its incident edges, is equivalent to adding an edge to the base graph, as shown in Figure 4.4. Let v be the vertex to be added to $L(G)$, and let X be the set of vertices to which v will be made adjacent. The endpoints of v (in the base) must cover X (as edges in the base), moreover, these endpoints must be incident with only these edges.

If $X = \emptyset$, then ADDVERTEX, the algorithm presented in Figure 4.6, creates two new vertices, b_1 and b_2, in the base, and puts v between them. Creating b_1 and b_2 takes O(1) time; however, placing v between b_1 and b_2 requires that we establish circular doubly linked lists for each of these vertices. Since each of these circular doubly linked lists contains only v, setting the *.ep*, *.nx*, *.prev*, and *.nn* values of v to represent the new circular doubly linked lists takes O(1) time.

If $X \neq \emptyset$, then we are looking for a set S of vertices in the base for which each of the following conditions hold.

- $1 \leq |S| \leq 2$.
- each edge of X (in the base) has exactly one endpoint in S.
- no edge of the base not in X has an endpoint in S.

We will call such a set S *valid*. This concept is illustrated in Figure 4.4.

To find a valid set, ADDVERTEX calls FINDVALID. FINDVALID selects an edge of the base, $edge_0$, from X and tries to include $edge_0.ep_0$ in a valid set. Letting X_0 be the subset of edges in X that are not incident with $edge_0.ep_0$, we observe that if we require another vertex in the valid set, then it must come from an edge in X_0. We initially set X_0 to X, then traverse the circular doubly linked list about $edge_0.ep_0$ to eliminate edges from X_0. If at any point we find an edge which does not belong to X, then $edge_0.ep_0$ cannot be in the valid set, so we backtrack and try $edge_0.ep_1$. If $edge_0.ep_1$ is similarly problematic, then the base will not yield a valid set. However, before concluding that v cannot be added to the line graph, we must determine if the component of the line graph containing $edge_0$ has another base which is not partition isomorphic. If so, we repeat our efforts on $edge_0$ using this new base.

Providing some endpoint of $edge_0$ can be added to the valid set, FINDVALID now selects an edge, $edge_1$, from X_0 and tries to include $edge_1.ep_0$ in the valid set. Letting X_1 be the subset of edges of X_0 that are not incident with $edge_1.ep_0$, we observe that $edge_1.ep_0$ can be added to complete the valid set if and only if all of the edges found in the circular doubly linked list about $edge_1.ep_0$ belong to X_0, and $X_1 = \emptyset$. We determine X_1 in a manner similar to that described for finding X_0 above, then backtrack if necessary. By backtracking, FindValid exhausts all combinations of bases and endpoints in finding a valid set. In particular, backtracking first tries a new endpoint, then, if necessary, a new base.

From Theorem 4.2, we see that any component of the line graph with two bases that are not partition isomorphic has O(1) vertices. Therefore, FindValid requires at most one base change, where each base change takes O(1) time. Moreover, for each possible base, there are at most four selections of $edge_i.ep_j$, where FindValid stops traversing the circular doubly linked list about $edge_i.ep_j$ as soon as it finds some vertex not in X/X_0. Therefore, FindValid takes $O(|X|)$ time.

If a valid set S is found, then we add v to the base graph using the vertices in S. If $S = \{s_1\}$, then AddVertex creates a new vertex b_1 (in G), which takes O(1) time, and places v between s_1 and b_1. Setting the $.ep$ values of v takes O(1) time and, as we have discussed, the creation of the circular doubly liked list about b_1 takes O(1) time. However, the addition of v to the circular doubly linked list, $\mathcal{L}_1$, about s_1 is more complicated. Inserting v into $\mathcal{L}_1$ after $edge_0$ takes only O(1) time, but we must also adjust the $.nn$ counters of every vertex in $\mathcal{L}_1$. Adjusting these counters takes $\Theta(|\mathcal{L}_1|) \in O(|X|)$ time.

If $S = \{s_1, s_2\}$, then AddVertex places v between s_1 and s_2. Again, setting the $.ep$ values of v takes O(1) time, and the addition of v to the circular doubly linked lists takes $\Theta(|\mathcal{L}_1| + |\mathcal{L}_2|)$ time, where $\mathcal{L}_i$ is the circular doubly linked list about s_i. However, $|\mathcal{L}_1| + |\mathcal{L}_2| = |X|$, so $\Theta(|\mathcal{L}_1| + |\mathcal{L}_2|) \in \Theta(|X|)$.

AddVertex($L(G), X$)

Input: An adjacency labelling of a line graph $L(G)$ (that is, the labels thereof) created using our dynamic scheme, and a subset X of $V_{L(G)}$. Note that the labels of $L(G)$ are only accessed as required.

Output: Let $L(G')$ be the graph formed by adding a new vertex v to $L(G)$, where v is adjacent to exactly those vertices in X. Providing $L(G')$ is a line graph, the output is an adjacency labelling of $L(G')$ (again, the labels thereof). If $L(G')$ is not a line graph, the output indicates as such.

```
create a new vertex $v$ (in $L(G)$)
if $X = \emptyset$ then
    create two new vertices $b_1$ and $b_2$ (in $G$)
    $v \leftarrow b_1 b_2$
else $S \leftarrow$ FindValid($X$)
    if $S = \emptyset$ then
        error no longer a line graph
    elseif $|S| = 1$ then
        create a new vertex $b_1$ (in $G$)
        $v \leftarrow s_1 b_1$
    else $v \leftarrow s_1 s_2$
```

Figure 4.6: The relabeller AddVertex which relabels the line graph when a vertex is added. The vertices s_1 and s_2, of the base, are the members of the valid set S

In total, AddVertex runs in $O(|X|)$ time. Given that AddVertex is input with the labels of each vertex in X, the running time of AddVertex is polynomial in the size of the input. Moreover, AddVertex is error-detecting since our use of backtracking guarantees that a valid set will be found, providing one exists.

Proposition 4.4 *The modification excess and modification locality of* AddVertex *are zero.*

Proof. First, observe that the set of vertices whose neighbourhoods change is $X \cup \{v\}$. If the label of a vertex x is modified, then its corresponding edge in the base had been in one of the circular linked lists about an endpoint of edge in the valid set. That is, $x \in X \cup \{v\}$. Therefore, the set of vertices with modified labels is a subset of the set of vertices whose neighbourhoods change, giving the desired result. ☐

Deleting an edge

Consider the act of deleting an edge from a line graph, as depicted in Figure 4.3. This deletion is equivalent to "pulling apart" two adjacent edges in the base. If there are additional edges incident with the vertex of the base at which these two adjacent edges were joined, then it becomes increasingly difficult to determine the new base graph. Fortunately, there are a finite number of cases to be considered; we enumerate these cases as a corollary of the following theorem.

Theorem 4.5 *Let $L(G)$ and $L(G')$ be line graphs, where $L(G') = L(G) - ab$. Moreover, let the edges of G corresponding to a and b be wx and wy, respectively. The following are properties of G.*

1. *The degree of w is at most four.*
2. *If $d = xy$ is an edge of G, then $deg(w) \leq 3$.*
3. *Let $c = wz$ be an edge of G, where $c \neq a, b$. If $f = zt$ is an edge of G, where $f \neq c$, then either $t = x$ or $t = y$.*
4. *If $d = xy$ and $c = wz$ are edges of G, where $c \neq a, b$, then there can be no edges adjacent to c other than a and b.*
5. *If $c = wz$ and $f = zx$ are edges of G, where $c \neq a, b$, then $deg(x) \leq 3$. Moreover, if $deg(x) = 3$, then x is adjacent to an edge $g = xp$, where $p \neq w, z, y$, such that $deg(p) \leq 2$. As well, if $deg(p) = 2$, then the edge $i = pw$ belongs to G.*
6. *Let $c = wz$ be an edge of G, where $c \neq a, b$. If there exist distinct edges $f = zt_1$ and $h = zt_2$, where $c \neq f, h$, then $\{t_1, t_2\} = \{x, y\}$. Moreover, $deg(x) = deg(y) = 2$.*

7. *If H is a subgraph of G, then $L(H) - ab$ is a line graph.*

Proof. Fundamental to the proofs of each of these observations is the fact that the only edge adjacencies that changes from G to G' is that of a and b which are adjacent in G but not in G'. To aid in the visualization of these proofs, the reader is encouraged to consult the diagrams in Table 4.1. It should be noted that, although there is a direct correspondence between edges of G and G', the same correspondence cannot be made between the vertices of G and G' as it is only the edge adjacencies which are important, not the specific vertices at which edges are adjacent. Consequently, any references to vertices in the following arguments will be in the context of the graph G.

1. Assume that w is incident with at least five edges in G, say a, b, c, i, and j. Now c, i, and j must be adjacent to both a and b in G' because they had been so in G. However, in a simple graph, any set of three edges between two disjoint edges, such as a and b in G', must also contain two disjoint edges. Without loss of generality, let these disjoint edges be c and j. This implies that c and j were not adjacent in G, which is a contradiction, as they were both incident with w.

2. Assume that w is incident with at least four edges in G, say a, b, c, and i. Now d must be adjacent to both a and b in G' because it had been so in G. Yet a and b are disjoint in G', so G' must contain the path adb of edges. Similarly, c must be adjacent to both a and b in G' because it had been so in G. Yet a and b are disjoint in G', so G' must contain the path acb of edges. Additionally, c must be disjoint from d in G' as it had been so in G; thereby, G' must contain the four cycle $adbc$ of edges. However, G' must contain the four cycle $adbi$ of edges as the arguments made for c can also be made for i. Thereby, $c = i$, which is a contradiction.

3. Using an argument identical to that made for the edge d in (2), G' must contain the path acb of edges. Yet f must be adjacent to c in G' because it had been so in G. Therefore, f is adjacent to at least one of a or b in both G' and G. Without loss of generality, let f be adjacent to a. If $t = w$, then $f = c$; therefore, $t = x$, as desired. This scenario is depicted in case E of Table 4.1.

4. By (2), the only edges adjacent to c at w (in G) are a and b, thereby, it remains to show that $deg(z) = 1$. Assume that there is another edge adjacent to c at z, namely $f = zt$. By (3), $t \in \{x, y\}$, so, without loss of generality, let $t = x$. Using the argument found in (2), we know that G' contains the four cycle $adbc$ of edges as shown in case B of Table 4.1. However, G' must also contain the four cycle $fdbc$ of edges, as the arguments made for a can also be made for f. Thereby, $f = a$, which is a contradiction.

5. Consider when $deg(x) > 2$, that is, there exists some edge $g = xp$ for which $p \notin \{w, z\}$, that is, $g \neq f, a$. From (4), we know that $p \neq y$, that is, $g \neq d$. Since $p \notin \{w, z\}$, g is not adjacent to c in G, nor in G'. Yet, g must be adjacent to both a and f in G' as it had been so in G; thereby, G' contains the thee cycle of edge afg, as shown in case F of Table 4.1. The uniqueness of this three cycle gives $deg(x) \leq 3$.

   Finally, consider when $deg(p) > 1$, that is, there exists some edge $i = ps$ for which $i \neq g$. Now i must be adjacent to g in G' as it had been so in G. But G' contains the three cycle afg of edges, as depicted in case F of Table 4.1, so, i must be adjacent to either a or f in G' and, subsequently, in G.

   If i is adjacent to f in G, then $i \neq g$ gives $s = z$; so, i is adjacent to c, f, and g, but neither a nor b in G and, subsequently, in G'. However, the configuration of G' depicted in case F of Table 4.1 requires that either $i = a$ or i is adjacent to b, both of which are contradictions. On the other hand, if i is adjacent to a in G, then $i \neq g$ gives $s = w$, as seen in case J of Table 4.1. Moreover, the uniqueness of the edge $i = pw$ gives $deg(p) \leq 2$.

6. The first part of this statement follows directly from (3), thereby, $deg(x), deg(y) \geq 2$. Without loss of generality, let $f = zx$ and let $h = zy$. Now f is adjacent to a and h, but not b, in G', as it had been so in G; similarly, h must be adjacent to b and f, but not a, in G', as it had been so in G. Thereby, G' must contain the path a, f, h, b of edges. As well, c must be adjacent to each of a, b, f, and h as it had been so in G, therefore, G' is as depicted in case G of Table 4.1. Note, in particular, that the edges c, f, and h form a three cycle in G'.

   Assume that $deg(x) > 2$. From (5), we know that x is incident with some edge $g = xp$, where $x \neq w, y, z$. Consequently, g is adjacent to f, but neither c nor h in G and, subsequently, G'. Yet, the edges c, f, and h form a triangle in G', so g cannot be adjacent to f in G' as it is adjacent to neither c nor h in G'. This contradiction gives $deg(x) \leq 2$. A similar argument gives that $deg(y) \leq 2$.

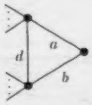
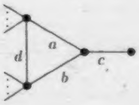
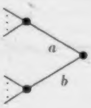
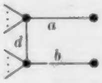
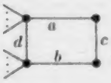
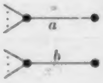
7. Given that H is a subgraph of G, $L(H)$ is an induced subgraph of $L(G)$. However, $L(G') = L(G) - ab$, so $L(H) - ab$ is an induced subgraph of $L(G')$. Since the family of line graphs is hereditary, $L(H) - ab$ is a line graph.

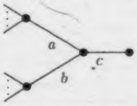
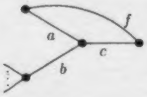
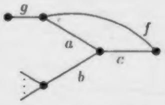
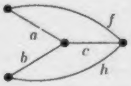
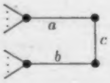
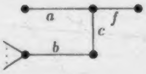
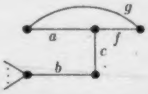
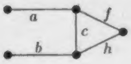
□

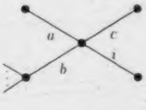
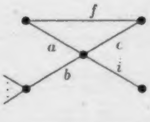
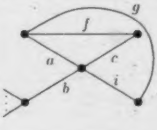
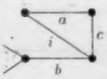
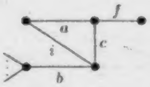
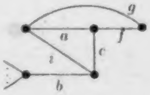
Corollary 4.6 *Let $L(G)$ and $L(G')$ be line graphs where $L(G') = L(G) - ab$. Table 4.1 classifies all of the possible base graphs, G, up to symmetry.*

Since the circular linked list structure distributes the information about the neighbourhood of a vertex across the labels of its neighbourhood, the vertex labels are sufficient to

Table 4.1: Possible cases for deleting an edge from (or adding an edge to) a line graph. In each case the edge *ab* is deleted from the line graph. The use of ellipses indicates that the graph extends arbitrarily from the indicated vertex

Case	A	B	C
G			
G′			

	D	E	F	G
G				
G′				

	H	I	J
G			
G′			

determine the local structures depicted in Table 4.1 (in fact, we can perform both depth first and breadth first search on the line graph and its base). Consequently, DeleteEdge, the algorithm presented in Figure 4.7, needs only identify if G has one of the structures shown in Table 4.1.

DeleteEdge($L(G), a, b$)

Input: An adjacency labelling of a line graph $L(G)$ (that is, the labels thereof) created using our dynamic scheme, and two distinct vertices a and b of $V_{L(G)}$ for which $ab \in E_{L(G)}$. Note that the labels of $L(G)$ are only accessed as required.

Output: An adjacency labeling of a graph $L(G')$ (again, the labels thereof) formed by deleting the edge ab from $L(G)$, providing $L(G')$ is a line graph. If $L(G')$ is not a line graph, then the output indicates as such.

```
examine the neighbourhood surrounding $a$ and $b$ (in $G$) to determine which of the cases in Table 4.1 applies to $G$
if none of the cases in Table 4.1 applies to $G$ then
    error no longer a line graph
else change the vertex labels of $a$, $b$, $c$, $d$, $f$, $g$, $h$, and $i$, as necessary, to reflect the new base graph $G'$
```

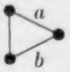
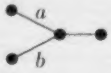
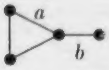
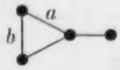
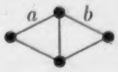
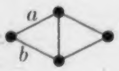
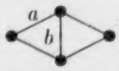
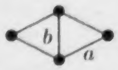
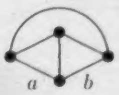
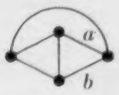
Figure 4.7: The relabeller DeleteEdge which relabels the line graph when an edge is deleted

Given that DeleteEdge is only concerned with the interaction between the edges a, b, c, d, f, g, h, and i, it can determine if G has one of the structures shown in Table 4.1 in O(1) time. If G does have one of the structures shown in Table 4.1, then DeleteEdge needs only to change the interaction between a, b, c, d, f, g, h, and i, as necessary, to reflect G'. From the cases shown in Table 4.1, observe that the change in the labels should not affect any endpoint of a or b that has arbitrary adjacency. Therefore, DeleteEdge can relabel the graph in O(1) time. As per our comments earlier in this section, the running time of DeleteEdge is polynomial in the size of the input.

Unlike the addition of a new vertex, the choice of base is irrelevant when it comes to deleting an edge from the line graph. Specifically, given a component of a line graph with two bases that are not partition isomorphic, if in one base the deleted edge leads to one of the configurations presented in Table 4.1, then so too will this edge in the other base. Therefore, given Corollary 4.6, DeleteEdge is error-detecting as it will exhaustively determine if G satisfies any of the cases shown in Table 4.1. In Table 4.2 we present all such pairs of bases that are not partition isomorphic in which the edge ab is to be deleted from the line graph, as well as the corresponding case of Table 4.1 to which each base belongs.

Proposition 4.7 *The modification excess and modification locality of* DeleteEdge *are four and one, respectively.*

Table 4.2: Pairs of partition non-isomorphic bases in which the edge *ab* is to be deleted from the line graph

Case in Table 4.1	Base 1	Base 2	Case in Table 4.1
A			D
E			A
G			A
none			none
none			none

Proof. First, observe that the set of vertices whose neighbourhoods change is $\{a, b\}$. From Table 4.1, we see that any edge whose endpoints change belongs to the set $\{a, b, c, d, f, g, h, i\}$. In particular, the largest such subset belongs to case J, where the set of edges with changed endpoints is $\{a, b, c, f, g, i\}$. Therefore, the modification excess is four. Moreover, each of the edges in $\{a, b, c, d, f, g, h, i\}$ corresponds to a vertex that is adjacent to either a or b in the line graph. Therefore, the modification locality of DELETEEDGE is one. □

Adding an edge

The act of adding an edge to a line graph is depicted in Figure 4.3. Since the process of adding an edge is exactly the reverse of deleting an edge, Table 4.1 enumerates all the possibilities.

Just as we saw with the deletion of an edge, the choice of base is irrelevant when it comes to adding a new edge to the line graph. Specifically, given a component of a line graph with two bases that are not partition isomorphic, if in one base the added edge leads to one of the configurations presented in Table 4.1, then so too will this edge in the other base. In Table 4.3 we present all such pairs of bases that are not partition isomorphic in which the edge ab is to be added to the line graph, as well as the corresponding case of Table 4.1 to which each base belongs.

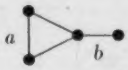
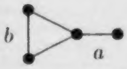
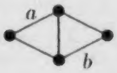
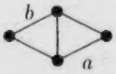
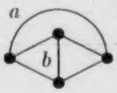
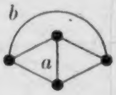
Again, the labels of the vertices in the line graph are sufficient to determine the local structures as depicted in Table 4.1 so the algorithm for updating the labels needs only identify the structure of the base, then alter the labels to represent the structure of the new base. Like DELETEEDGE, ADDEDGE runs in O(1) time, is error-detecting, has a modification excess of four, and has a modification locality of one.

4.3 Summary

In this chapter, we have developed an error-detecting dynamic adjacency labelling scheme for line graphs by using circular doubly linked lists to encode information about the base graph in the vertex labels. Our dynamic scheme for line graphs uses $O(\log n)$ bit labels and updates can be performed in $O(e)$ time, where e is the number of edges added to, or deleted from, the line graph.

In developing this dynamic scheme, we also defined the concept of partition isomorphism, and developed theory on the types of line graphs that can be modified to produce new line graphs.

Table 4.3: Pairs of partition non-isomorphic bases in which the edge *ab* is to be added to the line graph

Case in Table 4.1	Base 1	Base 2	Case in Table 4.1
H			H
none			none
none			none

Chapter 5

r-graphs

In this chapter, we develop error-detecting dynamic adjacency labelling schemes for classes of graphs defined using maximal cliques and maximal bicliques, namely, r-minoes and r-bics, respectively. Our interest in r-minoes and r-bics lies not so much with the classes themselves, rather, more with the maximal cliques and bicliques, which are structures commonly discussed in graph theory. A recent paper of Stix [54] offers a dynamic algorithm for maintaining maximal cliques in fuzzy clustering applications such as music and semantic clustering. A recent paper of Driskell, Ané, Burleigh, McMahon, O'Meara, and Sanderson [15] applies maximal bicliques to the field of genetics.

Metelsky and Tyshkevich [44] define a graph to be an *r-mino* if none of its vertices belongs to more than r maximal cliques. This notion of an r-mino is an extension of the idea of a domino, as defined by Kloks, Kratch, and Müller [36], in which each vertex belongs to at most two maximal cliques. In their work, Metelsky and Tyshkevich show that the class of r-minoes is the same as the class of line graphs of Helly hypergraphs with rank at most r; recall Definitions 2.6 and 2.7, regarding line graphs of hypergraphs. A hypergraph $H = (V, \mathcal{E})$ is said to satisfy the Helly property if every pairwise intersecting subset $\mathcal{E}'$ of $\mathcal{E}$ is such that $\bigcap_{e \in \mathcal{E}'} e \neq \emptyset$ [8].

Consistent with the definition of Prisner [48], a *biclique* is a complete bipartite vertex induced subgraph. We define a graph to be an *r-bic* if none of its vertices belongs to more than r maximal bicliques.

Using $O(r \log n)$ bit labels, our dynamic schemes allow the addition and deletion of vertices and edges. Observe that it is impractical to consider $r \in \Omega(n/\log n)$. As discussed in Section 2.1.1, a simple adjacency labelling scheme using $\Theta(n)$ bit labels can be developed from the rows of adjacency matrices; moreover, this scheme can be maintained dynamically. Given that our labels can use $\Theta(r \log n)$ bits, we cannot improve upon this simple scheme if $r \in \Omega(n/\log n)$.

For r-minoes, our relabeller handles edge addition and deletion in $O(r^2\mathbf{D})$ time, vertex addition in $O(r^2e^2)$ time, and vertex deletion in $O(r^2e)$ time, where $\mathbf{D}$ is the maximum degree of the vertices in the original graph, and e is the number of edges added to, or deleted from, the original graph. Unfortunately, if $r \in \omega(1)$, then our vertex deletion relabeller is not error-detecting. Similarly, if $r \in o(1)$, then our vertex addition relabeller is not error detecting.

For $r \in \Omega(1)$, our error-detecting vertex addition algorithm leads to an $O(r^2n^3)$ time recognition algorithm for r-minoes. This result offers an improvement over the $O(rn^4)$ algorithm that can be extended from work of Johnson, Yannakakis, and Papadimitriou [30], as well as the $O(n^{r+2})$ algorithm resulting from Metelsky and Tyshkevich's characterization using forbidden subgraphs [44].

For r-bics, our relabeller handles edge addition and deletion, as well as vertex deletion, in $O(r^2\mathbf{B})$ time, and vertex addition in $O(r^2n\mathbf{B})$ time, where $\mathbf{B}$ is the size of the largest biclique in the original graph. As with r-minoes, our error-detecting vertex addition algorithm leads to $O(r^2n^3)$ time recognition algorithm for r-bics. Unlike r-minoes, our relabeller will always be error-detecting.

5.1 The dynamic scheme for r-minoes

5.1.1 Vertex labels and decoder

Like the dynamic scheme for line graphs presented in Chapter 4, our dynamic adjacency labelling scheme for r-minoes uses graph substructures and circular doubly linked lists to distribute information about neighbourhoods across the vertices in the neighbourhoods. In the case of r-minoes, the important substructures are the maximal cliques. As such, we use the vertex labels to maintain a circular doubly linked list of the vertices in each maximal clique.

Given an r-mino on n vertices, each vertex is assigned a unique identifier; similarly, each maximal clique is assigned a unique identifier. For simplicity, we refer to vertices and maximal cliques by their identifiers. Given a vertex v, its label will also consist of the following information; exactly how the marker initially determines these labels will be addressed later.

$v.cin$: The number of maximal cliques in which v is contained.

$v.cl$: An array of triples with an entry for each maximal clique in which v is contained. Each member $v.cl_i$ is a triple of the form $(num, nx, prev)$, as follows, where the index i ranges from 1 to $v.cin$.

- num is the unique identifier assigned to the maximal clique.
- nx is a pair $(id, index)$, where id is the identifier of the next vertex after v in the circular doubly linked list of the vertices in maximal clique $v.cl_i.num$, and $index$

is the value j for which $id.cl_j.num = v.cl_i.num$.

- *prev* is a pair (*id*, *index*), where *id* is the identifier of the vertex before v in the circular doubly linked list of the vertices in maximal clique $v.cl_i.num$, and *index* is the value j for which $id.cl_j.num = v.cl_i.num$.

In particular, the label of a vertex is (v: $v.cin$; $v.cl$) as illustrated in Figure 5.1.

Given the labels of two vertices, v_1 and v_2, the decoder can determine the adjacency of v_1 and v_2 in $O(v_1.cin + v_2.cin) \in O(r)$ time, by comparing the $.num$ entries of $v_1.cl$ and $v_2.cl$ to see if the vertices share a common maximal clique. To this effect, let $\mathcal{C}_v$ denote $\{v.cl_i.num | 1 \leq i \leq v.cin\}$, the set of all maximal cliques containing v. We observe that $\mathcal{C}_v$ can be obtained in $\Theta(v.cin) \in O(r)$ time, where $|\mathcal{C}_v| = v.cin \leq r$. The vertices v_1 and v_2 are adjacent if and only if $\mathcal{C}_u \cap \mathcal{C}_v \neq \emptyset$.

To check the condition $\mathcal{C}_u \cap \mathcal{C}_v \neq \emptyset$, we use a reciprocal pointer technique suggested in a text by Aho, Hopcroft, and Ullman [2] (exercise 2.12). Consider two subsets S_1 and S_2 of S. To determine in $O(|S_1| + |S_2|)$ time if $S_1 \cap S_2 \neq \emptyset$, we require a block B_1 of memory to hold a stack of $|S_1|$ words, and a block B_2 of memory indexed by the elements of S. First, we initialize B_1 to 0. Then, for each element s_1^i of S_1, we push a pointer P_1^i onto the stack at B_1 and initialize a pointer P_2^i in position s_1^i of B_2, such that P_1^i points to P_2^i and vice versa. If, for some element s_2^j of S_2, position s_2^j of B_2 holds a pointer Q_2 that points to a pointer Q_1 in our stack, such that Q_1 points back to Q_2, then $S_1 \cap S_2 \neq \emptyset$. Observe that similar approaches can be used to determine $S_1 \cap S_2$ and $S_1 \cup S_2$ in $O(|S_1| + |S_2|)$ time as well.

Of particular interest is how we can use the vertex labels to traverse the circular doubly linked list of vertices in a maximal clique. For any maximal clique C, if we know some vertex s in C, as well as the value i for which $s.cl_i.num = C$, then the next vertex in the circular doubly linked list is $s.cl_i.nx.id$ and the $.cl$ entry of $s.cl_i.nx.id$ that corresponds to C is $s.cl_i.nx.index$. Moreover, $s.cl_i.nx.id$ and $s.cl_i.nx.index$ can be determined in $O(1)$ time. As such, the circular doubly linked list of vertices in C can be traversed in $O(|C|)$ time. For simplicity, we will say that we traverse C, although we really mean that we traverse the circular doubly linked list of vertices in C.

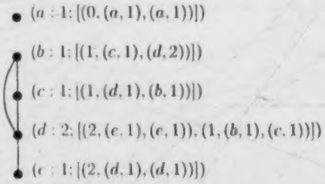

$(a : 1; [(0, (a, 1), (a, 1))])$
$(b : 1; [(1, (c, 1), (d, 2))])$
$(c : 1; [(1, (d, 1), (b, 1))])$
$(d : 2; [(2, (e, 1), (e, 1)), (1, (b, 1), (c, 1))])$
$(e : 1; [(2, (d, 1), (d, 1))])$

Figure 5.1: A labelling of 2-mino (domino) obtained using our labelling scheme

As well, we should address how we manage the *.cl* arrays. When a vertex v is added to a maximal clique, we simply create a new entry at the end of the array $v.cl$. When v is removed from a maximal clique, we delete the corresponding entry in $v.cl$. If a deleted entry had not been the last in the array, then we move the last entry to the position of the deleted entry; consequently, we must adjust the *.index* values for the previous and next vertices in the circular doubly linked list corresponding to the moved entry. In either case, we are able to maintain a contiguous array of $v.cin$ entries in O(1) time.

If $\overline{string}$ denotes the number of bits required to represent *string*, then the size of the label of v is

$$\overline{v}+\overline{v.cin}+\sum_{i=1}^{v.cin}\left(\overline{v.cl_i.num}+\overline{v.cl_i.prev.id}+\overline{v.cl_i.prev.index}+\overline{v.cl_i.nx.id}+\overline{v.cl_i.nx.index}\right).$$

Recall an earlier discussion in Section 3.1.3, where we observed that the dynamic nature of the graph might prevent the vertex and clique identifiers from being space-optimal. As such, let the largest identifier of a vertex in the graph be L_1, and let the largest identifier of a maximal clique be L_2. Thereby, $\overline{v}, \overline{v.cl_i.prev.id}, \overline{v.cl_i.nx.id} \in \mathrm{O}(\log L_1)$ and $\overline{v.cl_i.num} \in \mathrm{O}(\log L_2)$. Moreover, each vertex is in at most r maximal cliques, so $v.cin, v.cl_i.prev.index, v.cl_i.nx.index \leq r$, and the label of v uses $\mathrm{O}(\log L_1 + \log r + r\log L_2 + r\log r + r\log L_1) \in \mathrm{O}(r\log L_1 + r\log L_2)$ bits. If L_1 and L_2 are polynomial in n, which we stated as an assumption in Section 3.1.3, then the label size of v reduces to $\mathrm{O}(r\log n)$. In turn, the graph is represented using $\mathrm{O}(rn\log n)$ bits.

Using the same argument of Spinrad [53] (p. 18) found in Section 4.2.1, we can show that there are $2^{\Omega(n\log n)}$ labelled 1-minoes on n vertices. Yet, for $r' > r$, an r-mino is an r'-mino; thereby, there are $2^{\Omega(n\log n)}$ labelled r-minoes on n vertices. Therefore, our dynamic scheme for r-minoes is space-optimal when $r \in \mathrm{O}(1)$. For $r \in \omega(1)$, we cannot offer comment on optimality as we have no additional lower bounds. We also cannot offer comment on the optimality in the unlabelled case, when $r \in \mathrm{O}(1)$, as the $2^{\Omega(n\log n)}$ lower bound has not yet been established on the number of unlabelled r-minoes on n vertices.

5.1.2 Relabeller

Let us now examine the relabellers included in our dynamic scheme. As with the dynamic scheme presented for line graphs in Chapter 4, detailed pseudocode appears in Appendix C. In the following discussion, G is the original graph and G' is the changed graph.

Deleting a vertex.

Before we describe the relabeller, it is important to understand how the deletion of a vertex affects the maximal cliques in the graph. To this effect, consider the following lemma.

Lemma 5.1 *Consider a graph G' formed by deleting a vertex v from a graph G. Let $\mathcal{J}$ be the set of cliques $\{C|C$ is a maximal clique of G and $v \notin C\}$. The set of maximal cliques of G' can be partitioned as $\mathcal{J} \cup \{C \setminus \{v\}|C$ is a maximal clique of G, $v \in C$, and $C \setminus \{v\} \not\subseteq J$, for all $J \in \mathcal{J}\}$.*

Proof.

Consider an element J_0 of $\mathcal{J}$ and consider a set of the form $C \setminus \{v\}$ where C is a maximal clique of G, $v \in C$, and $C \setminus \{v\} \not\subseteq J$, for all $J \in \mathcal{J}$. By definition, $C \setminus \{v\} \neq J$, for all $J \in \mathcal{J}$; therefore, $C \setminus \{v\} \neq J_0$, which guarantees that the two sets of the claimed partition are disjoint.

Since J_0 is a clique of G, it is also a clique of G'. If J_0 is not a maximal clique of G', then there exists some clique C^* of G' for which $J_0 \subset C^*$. Since v is deleted from G, we know that $v \notin C^*$; thereby, C^* is a clique in G, which contradicts the maximality of J_0 in G. Therefore J_0 is a maximal clique of G'.

Since C is a clique in G, so too is $C \setminus \{v\}$ in G'. If $C \setminus \{v\}$ is not a maximal clique of G', then there exists some clique C^* of G' for which $C \setminus \{v\} \subset C^*$. Since v is deleted from G, we know that $v \notin C^*$; thereby, C^* is also a clique of G. Consequently, $C \setminus \{v\} \subset C^* \subseteq J^*$, for some J^* in $\mathcal{J}$, which is a contradiction. Therefore, $C \setminus \{v\}$ is a maximal clique of G'.

Having shown that the described sets are disjoint, and that their members are maximal cliques of G', it remains to show that these sets contain all the maximal cliques of G'. To this effect, let X denote the neighbourhood of v in G.

Consider a maximal clique C_0 of G'. If $C_0 \not\subseteq X$, then $C_0 \in \mathcal{J}$. On the other hand, if $C_0 \subseteq X$, then $C_0 \cup \{v\}$ is a maximal clique of G. Moreover, if there exists some J^* in $\mathcal{J}$, such that $C_0 \subseteq J^*$, then $C_0 \subset J^*$, as $J^* \not\subseteq X$ (otherwise, the clique $J^* \cup \{v\}$ would contradict the maximality of J^* in G). Yet, J^* is a clique of G', thereby, $C_0 \subset J^*$ contradicts the maximality of C_0 in G'. Therefore, for all J in $\mathcal{J}$, $C_0 \not\subseteq J$, as desired. □

Lemma 5.1, and its proof, suggest the design of our relabeller, DELETEVERTEX, as presented in Figure 5.2. All the maximal cliques of G that do not contain v will continue to be maximal in G'. However, for each maximal clique C of G, where $v \in C$, we must consider the possibility of $C \setminus \{v\}$ being contained in some maximal clique of $\mathcal{J}$.

The relabeller, DELETEVERTEX, first determines if v is an isolated vertex, which is to say that $v.cin = 1$ and $v.cl_1.nx = v$. If v is isolated, the algorithm frees the identifier of the maximal clique $\{v\}$ for future use. This case can be identified and addressed in O(1) time.

Providing v is not isolated, DELETEVERTEX obtains $\mathcal{C} = \mathcal{C}_v$, the set of maximal cliques containing v. As discussed, $\mathcal{C}_v$ can be determined in $\Theta(v.cin)$ time, where $|\mathcal{C}_v| = v.cin$. Recall that, to efficiently extract information about a maximal clique from the circular doubly linked list of its vertices, we must know the identifier of some vertex that belongs to

DeleteVertex(G, v)

Input: An adjacency labelling of an r-mino G (that is, the labels thereof) created using our dynamic scheme, and a vertex v in V_G. Note that the labels of G are only accessed as required.

Output: An adjacency labeling of a graph G' (again, the labels thereof) formed by deleting v from G, providing G' is an r-mino. If G' is not an r-mino, then the output indicates as such.

```
if $v$ is an isolated vertex then
    free the identifier of $\{v\}$
else $\mathcal{C} \leftarrow \{C | C$ is a maximal clique containing $v\}$
    for $C \in \mathcal{C}$ do
        if the only maximal clique of $G$ containing $C \setminus \{v\}$ is $C$ then
            remove $v$ from $C$
        else eliminate $C$
delete $v$ and free its identifier
```

Figure 5.2: The relabeller DeleteVertex which relabels an r-mino when a vertex is deleted

the maximal clique, as well as the index of the corresponding *.cl* entry. Consequently, for each entry $v.cl_i.num$ in $\mathcal{C}$, we assume that we also retain a reference to i.

For each maximal clique C of $\mathcal{C}$, the clique $C' = C \setminus \{v\}$ is examined to determine if it is maximal in G'. Where X denotes the neighbourhood of v in G, $|C| \leq |X| + 1$ and C' can be determined in $\Theta(|C|) \in O(|X|)$ time, by traversing C. Specifically, C' is maximal in G' if and only if the only maximal clique of G containing C' is C, which is to say that the set $\mathcal{A} = \bigcap_{c' \in C'} \mathcal{C}_{c'} \setminus C$ is empty. Each set $\mathcal{C}_{c'} \setminus C$ can be determined in $\Theta(c'.cin) \in O(r)$, so $\mathcal{A}$ can be determined in $\Theta(\sum_{c' \in C'} c'.cin) \in O(\sum_{x \in X} x.cin) \in O(r|X|)$ time, as per our discussion in Section 5.1.1.

If C' is maximal in G', then DeleteVertex merely removes v from C by removing v from the circular doubly linked list of vertices in C, eliminating the *.cl* entry of v that corresponds to C, and decrementing the $v.cin$ counter by one. This removal of v from C can be done in $O(1)$ time. On the other hand, if C' is not maximal in G' then DeleteVertex eliminates C by traversing C to decrease the *.cin* counters and delete the corresponding *.cl* entries, then freeing the identifier of C for future use. This elimination of C takes $\Theta(|C|) \in O(|X|)$ time.

Once all of the entries of $\mathcal{C}$ have been examined, DeleteVertex deletes v and frees its identifier for future use. So far, DeleteVertex has taken $O(v.cin \cdot \sum_{x \in X} x.cin) \in O(r^2|X|)$ time, where $\Theta(|X|)$ labels have been accessed. These $\Theta(|X|)$ vertex labels require $\Omega(v.cin + \sum_{x \in X} x.cin)$ bits, so the running time of DeleteVertex is polynomial in the size of its inputs.

If $r \in O(1)$, then DeleteVertex is error-detecting because the class is hereditary. However, if r is a strictly increasing function of n, then it is possible that the deletion of v

might cause another vertex to be in more than r_{n-1} maximal cliques.

Proposition 5.2 *The modification excess and modification locality of* DELETEVERTEX *are zero.*

Proof. First, observe that the set of vertices whose neighbourhoods change is $X \cup \{v\}$. If the label of a vertex x is modified, then x belongs to some maximal clique containing v. That is, $x \in X \cup \{v\}$. Therefore, the set of vertices with modified labels is a subset of the set of vertices whose neighbourhoods change, giving the desired result. □

Adding a vertex.

Consider the following lemma which describes how the addition of a vertex affects the maximal cliques in the graph.

Lemma 5.3 *Consider a graph G' formed by adding a vertex v to a graph G, where X denotes the neighbourhood of v. Let $\mathcal{I}$ be the set of cliques $\{C \cap X | C$ is a maximal clique of $G\}$. The set of maximal cliques of G' can be partitioned as $\{I \cup \{v\} | I$ is a maximal element of $\mathcal{I}\} \cup \{C | C$ is a maximal clique of G and $C \not\subseteq X\}$.*

Proof. Consider a set of the form $I \cup \{v\}$, where I is a maximal element of $\mathcal{I}$, and consider a set C, where C is a maximal clique of G and $C \not\subseteq X$. By definition, $v \in I \cup \{v\}$, however, v is not a vertex of G; therefore, $v \notin C$. Consequently, the two sets are disjoint.

Since I is a clique of G, where $I \subseteq X$, $I \cup \{v\}$ is a clique of G'. If $I \cup \{v\}$ is not a maximal clique of G', then there exists some clique C^* of G' for which $I \cup \{v\} \subset C^*$. Since $v \in C^*$ and C^* is a clique in G', we know that $C^* \setminus \{v\} \subseteq X$ and $I \subset C^* \setminus \{v\}$, which contradicts the maximality of I in $\mathcal{I}$. Therefore, $I \cup \{v\}$ is a maximal clique of G'.

Since C is a clique in G, it is also a clique in G'. If C is not a maximal clique of G', then there exists some clique C^* of G' for which $C \subset C^*$. However, $C \not\subseteq X$, therefore, $C^* \not\subseteq X$. If $C^* = \{v\}$, then $C = \emptyset$, which is a contradiction. On the other hand, if $C^* \neq \{v\}$, then $C^* \not\subseteq X$ gives $v \notin C^*$, as C^* is a clique. Since $v \notin C^*$, C^* is also a clique in G, which contradicts the maximality of C in G. Therefore, C is a maximal clique of G'.

Having shown that the described sets are disjoint, and that their members are maximal cliques of G', it remains to show that these sets contain all the maximal cliques of G'.

Consider a maximal clique C_0 of G'. If $v \notin C_0$, then C_0 is maximal in G. Moreover, $C_0 \not\subseteq X$, otherwise, the existence of the clique $C_0 \cup \{v\}$ in G' contradicts the maximality of C_0 in G'.

On the other hand, if $v \in C_0$, consider the clique $I_0 = C_0 \setminus \{v\}$ of G. Since $I_0 \subseteq X$, there must be some maximal element I^*, of $\mathcal{I}$, that contains I_0. If $I_0 \subset I^*$, then $C_0 = I_0 \cup \{v\} \subset I^* \cup \{v\}$, contradicting the maximality of C_0 in G'. Therefore, I_0 is a maximal element of $\mathcal{I}$, as desired. □

Lemma 5.3, and its proof, suggest the design of our relabeller, AddVertex, as presented in Figure 5.3. Specifically, the maximal cliques in G' of the form $(C \cap X) \cup \{v\}$, where C is a maximal clique of G and $C \subseteq X$, are achieved by adding v to C. On the other hand, those maximal cliques of the form $(C \cap X) \cup \{v\}$, where C is a maximal clique of G and $C \not\subseteq X$, are achieved by creating an entirely new maximal clique. By creating new maximal cliques in these situations, we carry forward all those maximal cliques of the form C, where C is a maximal clique of G and $C \not\subseteq X$.

The relabeller, AddVertex, first establishes the new vertex v. It does this in $O(1)$ time by assigning an identifier to v and setting $v.cin$ to 0. If $X = \emptyset$, then v is an isolated vertex; therefore, the only new maximal clique in G' is $\{v\}$. How AddVertex creates new maximal cliques is discussed shortly.

Providing $X \neq \emptyset$, AddVertex first obtains $\mathcal{C} = \cup_{x \in X} \mathcal{C}_x$, the set of maximal cliques that contain some member of X. The set $\mathcal{C}$ contains no more than $\sum_{x \in X} x.cin$ entries. Since $x.cin \leq r$, $\sum_{x \in X} x.cin \leq r|X|$. Moreover, $\mathcal{C}$ can be determined in $\Theta(\sum_{x \in X} x.cin) \in O(r|X|)$ time using a reciprocal pointer technique like that presented in Section 5.1.1. Recall that to efficiently extract information about a maximal clique from the circular doubly linked list of its vertices we must know the identifier of some vertex that belongs to the maximal clique, as well as the index of the corresponding $.cl$ entry. Consequently, for each entry $x.cl_i.num$ in $\mathcal{C}$, we assume that we also retain a reference to x and i.

For each maximal clique C in $\mathcal{C}$, we entertain the possibility of $C' = C \cap X$ being a maximal element of the set $\mathcal{I}$, seen in Lemma 5.3. As discussed in Section 5.1.1, $C \cap X$ can be computed in $O(|C| + |X|)$ time, however, $|C|$ could be as large as n; as such, we prefer to determine $C \cap X$ in $\Theta(\sum_{x \in X} x.cin) \in O(r|X|)$ time by searching for the identifier of C in the $.num$ entries of each element of X. If C' is a subclique of some maximal clique C^* remaining in $\mathcal{C}$ (observe that AddVertex removes C from $\mathcal{C}$ when it is selected), then we do nothing as $C^* \cap X$ will present itself later. Specifically, if $C' \subset C^* \cap X$, then C' is not a maximal element of $\mathcal{I}$; otherwise, if $C' = C^* \cap X$, then we avoid possible duplication of maximal cliques. Similarly, if C' is a subclique of some maximal clique C^* in $\mathcal{D}$, the set of maximal cliques D for which $(D \cap X) \cup \{v\}$ has been made a maximal clique of G', then $C' \subset C^* \cap X$ (given that C^* is selected from $\mathcal{C}$ before C', the addition of C^* to $\mathcal{D}$ was contingent on $C^* \cap X \not\subseteq C'$).

The clique C' is a subclique of some maximal clique remaining in $\mathcal{C} \cup \mathcal{D}$ if and only if $(\cap_{c' \in C'} \mathcal{C}_{c'}) \cap (\mathcal{C} \cup \mathcal{D}) \neq \emptyset$. Since no vertex is contained in more than r maximal cliques, $\cap_{c' \in C'} \mathcal{C}_{c'}$ can be computed in $\Theta(\sum_{c' \in C'} c'.cin) \in O(\sum_{x \in X} x.cin) \in O(r|C'|) \in O(r|X|)$ time. In turn, the condition $(\cap_{c' \in C'} \mathcal{C}_{c'}) \cap (\mathcal{C} \cup \mathcal{D}) \neq \emptyset$ can be checked in $O(\sum_{x \in X} x.cin) \in O(r|X|)$ time, as $|\mathcal{D}| \leq |\mathcal{C}| \leq \sum_{x \in X} x.cin \leq r|X|$.

AddVertex(G, X)

Input: An adjacency labelling of an r-mino G (that is, the labels thereof) created using our dynamic scheme, and a subset X of V_G. Note that the labels of G are only accessed as required.

Output: Let G' be the graph formed by adding a new vertex v to G, where v is adjacent to exactly those vertices in X. Providing G' is an r-mino, the output is an adjacency labelling of G' (again, the labels thereof). If G' is not an r-mino, the output indicates as such.

```
create a new vertex $v$
if $X = \emptyset$ then
    make a new maximal clique $\{v\}$
else $\mathcal{C} \leftarrow \{C | C$ is a maximal clique containing a vertex of $X\}$
    $\mathcal{D} \leftarrow \emptyset$
    for $C \in \mathcal{C}$ do
        $\mathcal{C} \leftarrow \mathcal{C} \setminus \{C\}$
        if $C \cap X$ is not contained in any of the maximal cliques in $\mathcal{C}$ or $\mathcal{D}$ then
            $\mathcal{D} \leftarrow \mathcal{D} \cup \{C\}$
            if $C \cap X = C$ then
                add $v$ to $C$
            else make a new maximal clique $(C \cap X) \cup \{v\}$
```

Figure 5.3: The relabeller AddVertex which relabels an r-mino when a vertex is added

Now, if C' is not contained in any maximal clique of $\mathcal{C} \cup \mathcal{D}$, then we make a maximal clique of $C' \cup \{v\}$. Exactly how we make a maximal clique of $C' \cup \{v\}$ depends on whether $C' = C$, that is, whether $C \subseteq X$; note that this condition can be checked while computing C'. If $C' = C$, then we simply add v to C, as C will no longer be maximal in G'. Specifically, AddVertex inserts v into the circular doubly linked list for C, then increases the value of $v.cin$ by 1. This can be done in O(1) time. Otherwise, if $C' \neq C$, then we form a new maximal clique, $C' \cup \{v\}$, as C will continue to be maximal in G'. Specifically, AddVertex increases the value of the $.cin$ counter of each of the vertices in $C' \cup \{v\}$ by one, establishes a circular doubly linked list of the vertices in $C' \cup \{v\}$, and creates a new $.cl$ entry for each member of $C' \cup \{v\}$ while establishing the circular doubly linked list; forming this new maximal clique takes $O(|C'|) \in O(|X|)$ time. Whenever a $.cin$ counter is increased, we check that its value is no greater than r. Therefore, AddVertex is error-detecting, providing $r \in \Omega(1)$.

Having carefully examined AddVertex, we observe that the algorithm runs in $O((\sum_{x \in X} x.cin)^2) \in O(r^2|X|^2) \in O(r^2n^2)$ time, where $\Theta(|X|)$ vertex labels have been accessed. These $\Theta(|X|)$ vertex labels require $\Omega(v.cin + \sum_{x \in X} x.cin)$ bits, so the running time of AddVertex is polynomial in the size of its input.

Proposition 5.4 *The modification excess and modification locality of* AddVertex *are zero.*

Proof. First, observe that the set of vertices whose neighbourhoods change is $X \cup \{v\}$. If the label of a vertex x is modified, then x belongs to some maximal clique containing v. That is, $x \in X \cup \{v\}$. Therefore, the set of vertices with modified labels is a subset of the set of vertices whose neighbourhoods change, giving the desired result. ☐

Given Corollary 3.8, AddVertex gives polynomial time recognition for r-minoes.

Theorem 5.5 *For any graph G on n vertices, we can determine if G is an r-mino in $O(r^2n^3)$ time.*

Furthermore, observe that we can use AddVertex, just as we did in the proof of Theorem 3.6, to create a $O(r^2n^3)$ time marker for our dynamic scheme.

Deleting an edge.

Consider the following lemma, which describes how the deletion of an edge affects the maximal cliques in the graph.

Lemma 5.6 *Consider a graph G' formed by deleting an edge uv from a graph G. Let $\mathcal{L}$ be the set of cliques $\{C | C$ is a maximal clique of G, and $v \notin C$ or $u \notin C\}$. The set of maximal cliques of G' can be partitioned as $\mathcal{L} \cup \{C \setminus \{u\} | C$ is a maximal clique of G, $u, v \in C$, and $C \setminus \{u\} \not\subseteq L$, for all $L \in \mathcal{L}\} \cup \{C \setminus \{v\} | C$ is a maximal clique of G, $u, v \in C$, and $C \setminus \{v\} \not\subseteq L$, for all $L \in \mathcal{L}\}$.*

At the heart of Lemma 5.6 is the fact that the maximal cliques of G that do not contain both u and v will continue to be maximal in G'. However, for each maximal clique C of G containing both u and v, we must consider the possibility of $C \setminus \{u\}$ or $C \setminus \{v\}$ being contained in some other maximal clique of $\mathcal{L}$.

Proof. Consider an element L_0 of $\mathcal{L}$, a set of the form $C_u \setminus \{u\}$, where C_u is a maximal clique of G, $u, v \in C_u$, and $C_u \setminus \{u\} \not\subseteq L$, for all $L \in \mathcal{L}$, and a set of the form $C \setminus \{v\}$, where C_v is a maximal clique of G, $u, v \in C_v$, and $C_v \setminus \{v\} \not\subseteq L$, for all $L \in \mathcal{L}$. By definition, for all $L \in \mathcal{L}$, $C_u \setminus \{u\}, C_v \setminus \{v\} \neq L$; moreover, $v \in C_u \setminus \{u\}$ and $u \in C_v \setminus \{v\}$, so the three sets are disjoint.

Since L_0 is a clique of G, it is also a clique of G'. If L_0 is not a maximal clique of G', then there exists some clique C^* of G' for which $L_0 \subset C^*$. Since C^* is a clique, at least one of u and v does not belong to C^* as the edge uv does not belong to G'. Thereby, C^* is a clique in G, which contradicts the maximality of L_0 in G. Therefore, L_0 is a maximal clique of G'.

Since C_u is a clique, so too is $C_u \setminus \{u\}$ in G'. If $C_u \setminus \{u\}$ is not a maximal clique of G', then there exists some clique C^* of G' for which $C_u \setminus \{u\} \subset C^*$. Since C^* is a clique, at least one of u and v does not belong to C^* as the edge uv does not belong to G'; thereby,

C^* is also a clique of G. Consequently, $C_u \setminus \{u\} \subset C^* \subset J^*$, for some J^* in $\mathcal{J}$, which is a contradiction. Therefore, $C_u \setminus \{u\}$ is a maximal clique of G'. A similar argument gives that $C_v \setminus \{v\}$ is a maximal clique of G'.

Having shown that the described sets are disjoint, and that their members are maximal cliques of G', it remains to show that these sets contain all the maximal cliques of G'.

Consider a maximal clique C_0 of G'. If $u, v \notin C_0$, then $C_0 \in \mathcal{L}$. Similarly, if $u \in C_0$, $v \notin C_0$, and $C_0 \not\subseteq X_v$, then $C_0 \in \mathcal{L}$.

If $u \in C_0$, $v \notin C_0$, and $C_0 \subseteq X_v$, then $C_0 \cup \{v\}$ is a maximal clique of G. Moreover, if there exists some L^* in $\mathcal{L}$, such that $C_0 \subseteq L^*$, then $C_0 \subset L^*$, as $L^* \not\subseteq X_v$ (otherwise, the clique $L^* \cup \{v\}$ would contradict the maximality of L^* in G). Yet, L^* is a clique of G', thereby, $C_0 \subset L^*$ contradicts the maximality of C_0 in G'. Therefore, for all L in $\mathcal{L}$, $C_0 \not\subseteq L$, as desired.

When $v \in C_0$ and $u \notin C_0$, we can use similar arguments to show that C_0 has one of the desired forms. □

Lemma 5.6, and its proof, suggest the design of our relabeller, DELETEEDGE, as presented in Figure 5.4. Just as Lemma 5.6 resembles Lemma 5.1, so too does DELETEEDGE resemble DELETEVERTEX.

The relabeller, DELETEEDGE, first obtains $\mathcal{C} = \mathcal{C}_u \cap \mathcal{C}_v$, the set of maximal cliques containing both u and v, where $\mathcal{C}_u$ and $\mathcal{C}_v$ can be determined in $\Theta(u.cin)$ and $\Theta(v.cin)$ time, respectively. Since $|\mathcal{C}_u| = u.cin$ and $|\mathcal{C}_v| = v.cin$, $\mathcal{C}$ can be determined in $O(u.cin + v.cin) \in O(r)$ time. For each maximal clique C of $\mathcal{C}$, the cliques $C'_u = C \setminus \{u\}$ and $C'_v = C \setminus \{v\}$ are examined to determine if they maximal in G'. Letting X denote the intersection of the neighbourhoods of u and v in G, $|C| \leq |X| + 2$, so C'_u and C'_v can be determined in $O(|C|) \in O(|X|)$ time, by traversing C. For $\alpha \in \{u, v\}$, C'_α is maximal in G' if and only if the set $\mathcal{A}_\alpha = \bigcap_{c' \in C'_\alpha} (\mathcal{C}_{c'} \setminus C)$ is empty. As per our discussion in Section 5.1.1, $\mathcal{A}_\alpha$ can be determined in $O(\sum_{c' \in C'_\alpha} c'.cin) \in O(\sum_{x \in X} x.cin) \in O(r|X|)$ time.

First, let us assume that exactly one of $C \setminus \{u\}$ and $C \setminus \{v\}$ is a maximal clique in G'; without loss of generality, let the maximal clique be $C \setminus \{v\}$. We develop this maximal clique by removing v from C, just as we did in DELETEVERTEX, using $O(1)$ time. As a second possibility, consider if neither $C \setminus \{u\}$ nor $C \setminus \{v\}$ is a maximal clique in G'. In this case, we eliminate the maximal clique C, just as we did in DELETEVERTEX, using $O(|C|) \in O(|X|)$ time. Finally, if both $C \setminus \{u\}$ and $C \setminus \{v\}$ are both maximal in G', then we develop $C \setminus \{v\}$ by removing v from C, using $O(1)$ time; however, $C \setminus \{u\}$ must be developed by establishing a new maximal clique, just as we did in ADDVERTEX, requiring $O(|C'_u|) \in O(|X|)$ time.

Whenever a *.cin* counter is increased during the creation of the new maximal clique, we check that its value is no greater than r. Thereby, DELETEEDGE is error-detecting. Since $\mathcal{C}$ contains at most $\min\{u.cin, v.cin\}$ maximal cliques, the total running time of DELETEEDGE

DELETEEDGE(G, u, v)

Input: An adjacency labelling of an r-mino G (that is, the labels thereof) created using our dynamic scheme, and two distinct vertices u and v of V_G for which $uv \in E_G$. Note that the labels of G are accessed only as required.

Output: An adjacency labeling of a graph G' (again, the labels thereof) formed by deleting the edge uv from G, providing G' is an r-mino. If G' is not an r-mino, then the output indicates as such.

```
$\mathcal{C} \leftarrow \{C | C$ is a maximal clique containing both $u$ and $v\}$
for $C \in \mathcal{C}$ do
    if the only maximal clique of $G$ containing $C \setminus \{v\}$ is $C$ then
        remove $v$ from $C$
        if the only maximal clique of $G$ containing $C \setminus \{u\}$ is $C$ then
            make a new maximal clique of $C \setminus \{u\}$
    elseif the only maximal clique of $G$ containing $C \setminus \{u\}$ is $C$ then
        remove $u$ from $C$
    else eliminate $C$
```

Figure 5.4: The relabeller DELETEEDGE which relabels an r-mino when an edge is deleted

is $O(\min\{u.cin, v.cin\} \cdot \sum_{x \in X} x.cin) \in O(r^2|X|)$, where $\Theta(|X|)$ labels are accessed. Given that these labels require $\Omega(u.cin + v.cin + \sum_{x \in X} x.cin)$ bits, the running time of DELETEEDGE is polynomial in the size of its input.

Proposition 5.7 *The modification locality of* DELETEEDGE *is one.*

Proof. First, observe that the set of vertices whose neighbourhoods change is $\{u, v\}$. If the label of a vertex x is modified, then x belongs to some maximal clique of G containing u and v, giving the desired result. □

Adding an edge.

Consider the following lemma, which describes how the addition of an edge affects the maximal cliques in the graph.

Lemma 5.8 *Consider a graph G' formed by adding an edge uv to a graph G. Let X denote the neighbourhood of v in G' and let $\mathcal{K}$ be the set of cliques $\{C \cap X | C$ is a maximal clique of G and $u \in C\}$. The set of maximal cliques of G' can be partitioned as $\{K \cup \{v\} | K$ is a maximal element of $\mathcal{K}\} \cup \{C | C$ is a maximal clique of G, $C \not\subseteq X$, and $C \neq \{v\}\}$.*

Proof. Consider a set of the form $K \cup \{v\}$, where K is a maximal element of $\mathcal{K}$, and consider a set C, where C is a maximal clique of G, $C \not\subseteq X$, and $C \neq \{v\}$. By definition, $u, v \in K \cup \{v\}$; however, if $u \in C$, then $v \notin C$, as the edge uv does not belong to G. Therefore, the two sets are disjoint.

Since K is a clique of G, where $K \subseteq X$, $K \cup \{v\}$ is a clique of G'. If $K \cup \{v\}$ is not a maximal clique of G', then there exists some clique C^* of G' for which $K \cup \{v\} \subset C^*$. Since

$v \in C^*$ and C^* is a clique in G', we know that $C^* \setminus \{v\} \subseteq X$ and $K \subset C^* \setminus \{v\}$, which contradicts the maximality of K in $\mathcal{K}$. Therefore, $K \cup \{v\}$ is a maximal clique of G'.

Since C is a clique in G, it is also a clique in G'. If C is not a maximal clique of G', then there exists some clique C^* of G' for which $C \subset C^*$. However, $C \not\subseteq X$, therefore, $C^* \not\subseteq X$. If $C^* = \{v\}$, then $C = \emptyset$, which is a contradiction. On the other hand, if $C^* \neq \{v\}$, then $C^* \not\subseteq X$ gives $v \notin C^*$ as C^* is a clique. Since $v \notin C^*$, C^* is also a clique in G, which contradicts the maximality of C in G. Therefore, C is a maximal clique of G'.

Having shown that the described sets are disjoint, and that their members are maximal cliques of G', it remains to show that these sets contain all the maximal cliques of G'.

Consider a maximal clique C_0 of G'. If $u \notin C_0$ or $v \notin C_0$, then C_0 is maximal in G. Moreover, $C_0 \not\subseteq X$, otherwise, the existence of the clique $C_0 \cup \{v\}$ in G' contradicts the maximality of C_0 in G'. Similarly, $C_0 \neq \{v\}$, otherwise, the existence of the clique $C_0 \cup \{u\}$ in G' contradicts the maximality of C_0 in G'.

On the other hand, if $u, v \in C_0$, consider the clique $K_0 = C_0 \setminus \{v\}$ of G. Since $K_0 \subseteq X$, there must be some maximal element K^*, of $\mathcal{K}$, that contains K_0. If $K_0 \subset K^*$, then $C_0 = K_0 \cup \{v\} \subset K^* \cup \{v\}$, contradicting the maximality of C_0 in G'. Therefore, K_0 is a maximal element of $\mathcal{K}$, as desired. □

When forming a maximal clique of G' that contains both u and v, we can view it as either adding v to some clique containing u, or vice versa. In designing our relabeller, we have chosen the former viewpoint. The relabeller, AddEdge, presented in Figure 5.5, closely resembles AddVertex, just as Lemma 5.8 closely resembles Lemma 5.3.

The relabeller, AddEdge, first determines if v is an isolated vertex in G. If v is isolated, then the algorithm eliminates the maximal clique $\{v\}$, as we will later include v in some maximal clique containing u. As discussed in Section 5.1.2, this case can be identified and addressed in O(1) time.

Providing v is not an isolated vertex, AddEdge obtains X, the neighbourhood of v in G'. The neighbourhood of v in G can be determined by traversing each of the maximal cliques in $\mathcal{C}_v$. Since no vertex in G belongs to more than r maximal cliques, where the size of the largest maximal clique containing v is $|X| - 1$, the neighbourhood of v in G can be determined in $O(v.cin \cdot |X|) \in O(r|X|)$ time. By adding u to the neighbourhood of v in G, we obtain X.

As well, AddEdge obtains $\mathcal{C} = \mathcal{C}_u$, the set of maximal cliques containing u; again, for each element $u.cl_i.num$, we retain a reference to i. For each maximal clique C in $\mathcal{C}$, we process the subclique $C' = C \cap X$ as we did in AddVertex. In AddVertex, $|\mathcal{C}| \leq r|X|$, however, in AddEdge, $|\mathcal{C}| = u.cin \leq r$. Therefore, the running time of AddEdge is $O(u.cin \cdot v.cin \cdot |X|) \in O(r^2|X|)$, where $\Theta(|X|)$ labels are accessed. Given that these labels require $\Omega(u.cin + v.cin + \sum_{x \in X} x.cin)$ bits, the running time of AddEdge is polynomial in

```
ADDEDGE($G, u, v$)
Input: An adjacency labelling of an $r$-mino $G$ (that is, the labels thereof) created using our
dynamic scheme, and two distinct vertices $u$ and $v$ of $V_G$ for which $uv \notin E_G$. Note that the
labels of $G$ are only accessed as required.
Output: An adjacency labeling of a graph $G'$ (again, the labels thereof) formed by adding
the edge $uv$ to $G$, providing $G'$ is an $r$-mino. If $G'$ is not an $r$-mino, then the output indicates
as such.

if $v$ is an isolated vertex (in $G$) then
    eliminate the maximal clique $\{v\}$
$X \leftarrow \{x | x$ is a neighbour of $v$ (in $G'$)$\}$
$\mathcal{C} \leftarrow \{C | C$ is a maximal clique containing $u\}$
$\mathcal{D} \leftarrow \emptyset$
for $C \in \mathcal{C}$ do
    $\mathcal{C} \leftarrow \mathcal{C} \setminus \{C\}$
    if $C \cap X$ is not contained in any of the maximal cliques in $\mathcal{C}$ or $\mathcal{D}$ then
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{C\}$
        if $C \cap X = C$ then
            add $v$ to $C$
        else make a new maximal clique $(C \cap X) \cup \{v\}$
```

Figure 5.5: The relabeller ADDEDGE which relabels an r-mino when an edge is added

the size of its input. Just as we did in ADDVERTEX, whenever a vertex is added to a new maximal clique we check that it does not belong to more than r maximal cliques. Therefore, ADDEDGE is error-detecting.

Proposition 5.9 *The modification locality of* ADDEDGE *is one.*

Proof. First, observe that the set of vertices whose neighbourhoods change is $\{u, v\}$. If the label of a vertex x is modified, then x will belong to some maximal clique of G' containing v, giving the desired result. □

5.2 The dynamic scheme for r-bics

The dynamic adjacency labelling scheme that we develop for r-bics will be very similar to the one previously developed for r-minoes in Section 5.1. Before presenting our work on r-bics we note that the class of r-minoes does not include the class of r-bics, and vice versa. For example, $K_{1,3}$ is a 1-bic, but not a 1-mino, and K_3 is a 1-mino, but not a 1-bic.

5.2.1 Vertex labels and decoder

Like the dynamic scheme for r-minoes presented in Section 5.1, our dynamic adjacency labelling scheme for r-bics uses graph substructures and circular doubly linked lists to distribute information about neighbourhoods across the vertices in the neighbourhoods. In the case of r-bics, the important substructures are the maximal bicliques. As such, we use the vertex labels to maintain a circular doubly linked list of the vertices in each maximal biclique.

Given an r-bic on n vertices, each vertex is assigned a unique identifier, similarly, each maximal biclique is assigned a unique identifier. For simplicity, we refer to vertices and maximal bicliques by their identifiers. Given a vertex v, its label will also consist of the following information; exactly how the marker initially determines these labels will be addressed later.

$v.bin$: The number of maximal bicliques in which v is contained.

$v.bicl$: An array of 4-tuples with an entry for each maximal biclique in which v is contained. Each member $v.cl_i$ is a 4-tuple of the form $(num, part, nx, prev)$, as follows, where the index i ranges from 1 to $v.bin$.

- *num* is the unique identifier assigned to the maximal biclique.
- *part* is a value, either 0 or 1, used to indicate the part of the bipartition to which v belongs.
- *nx* is a pair (*id*, *index*), where *id* is the identifier of the next vertex after v in the circular doubly linked list of the vertices in maximal biclique $v.bicl_i.num$, and *index* is the value j for which $id.bicl_j.num = v.bicl_i.num$.
- *prev* is a pair (*id*, *index*), where *id* is the identifier of the vertex before v in the circular doubly linked list of the vertices in maximal biclique $v.bicl_i.num$, and *index* is the value j for which $id.bicl_j.num = v.bicl_i.num$.

In particular, the label of a vertex is (v: $v.bin$; $v.bicl$) as illustrated in Figure 5.6.

Given the labels of two vertices, v_1 and v_2, the decoder can determine the adjacency of v_1 and v_2 by comparing the *.num* and *.part* entries of $v_1.bicl$ and $v_2.bicl$ to see if the vertices belong to distinct parts of some common maximal biclique. To this effect, let $\mathcal{B}_v$ denote $\{v.bicl_j.num | 1 \leq j \leq v.bin\}$, the set of maximal bicliques containing v, and, for $i \in \{0, 1\}$, let $\mathcal{B}_v^i$ denote $\{v.bicl_j.num | 1 \leq j \leq v.bin, v.bicl_j.part = i\}$, the set of maximal bicliques containing v, where v belongs to the i^{th} part of the bipartition. Clearly, $\mathcal{B}_v^0$ and $\mathcal{B}_v^1$ partition $\mathcal{B}_v$, where $\mathcal{B}_v = v.bin$.

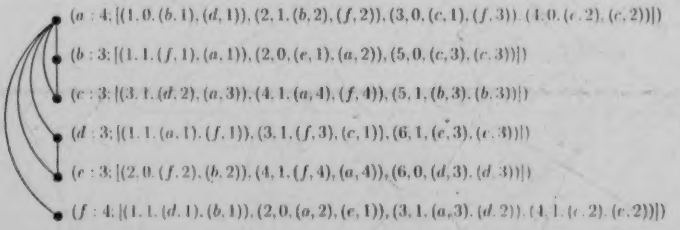

(a : 4; [(1, 0, (b, 1), (d, 1)), (2, 1, (b, 2), (f, 2)), (3, 0, (c, 1), (f, 3)), (4, 0, (e, 2), (c, 2))])
(b : 3; [(1, 1, (f, 1), (a, 1)), (2, 0, (e, 1), (a, 2)), (5, 0, (c, 3), (c, 3))])
(c : 3; [(3, 1, (d, 2), (a, 3)), (4, 1, (a, 4), (f, 4)), (5, 1, (b, 3), (b, 3))])
(d : 3; [(1, 1, (a, 1), (f, 1)), (3, 1, (f, 3), (c, 1)), (6, 1, (e, 3), (e, 3))])
(e : 3; [(2, 0, (f, 2), (b, 2)), (4, 1, (f, 4), (a, 4)), (6, 0, (d, 3), (d, 3))])
(f : 4; [(1, 1, (d, 1), (b, 1)), (2, 0, (a, 2), (e, 1)), (3, 1, (a, 3), (d, 2)), (4, 1, (e, 2), (c, 2))])

Figure 5.6: A labelling of a 4-bic obtained using our labelling scheme

For any vertex v, $\mathcal{B}_v$, $\mathcal{B}_v^0$, and $\mathcal{B}_v^1$, can be determined simultaneously in $\Theta(v.bin) \in O(r)$ time. Therefore, the adjacency of two vertices v_1 and v_2 can be determined in $O(v_1.bin + v_2.bin) \in O(r)$ time by testing the condition $(B_{v_1}^0 \cap B_{v_2}^1) \cup (B_{v_1}^1 \cap B_{v_2}^0) \neq \emptyset$, using the reciprocal pointer technique described in Section 5.1.1.

Using an approach identical to that seen in Section 5.1.1 for r-minoes, we are able to traverse the circular doubly linked list of vertices in a maximal biclique B in $O(|B|)$ time (again, we will say that we traverse B, although we really mean that we traverse the circular doubly linked list of vertices in B). While traversing B, we are also able to determine B_i, the vertices of B belonging to part i, where $i \in \{0, 1\}$. As well, we can use the approach seen in Section 5.1.1 to maintain $v.bicl$ as a contiguous array of $v.bin$ entries in $O(1)$ time.

If $\overline{string}$ denotes the number of bits required to represent $string$, then the size of the label of v is

$$\overline{v} + \overline{v.bin} + \sum_{i=1}^{v.cin} \Big(\overline{v.bicl_i.num} + \overline{v.bicl_i.part} + \\ \overline{v.bicl_i.prev.id} + \overline{v.bicl_i.prev.index} + \overline{v.bicl_i.nx.id} + \overline{v.bicl_i.nx.index}\Big).$$

Recall an earlier discussion in Section 3.1.3, where we observed that the dynamic nature of the graph might prevent the vertex and biclique identifiers from being space-optimal. As such, let the largest identifier of a vertex in the graph be L_1, and let the largest identifier of a maximal biclique be L_2. Thereby, $\overline{v}, \overline{v.bicl_i.prev.id}, \overline{v.bicl_i.nx.id} \in O(\log L_1)$ and $\overline{v.bicl_i.num} \in O(\log L_2)$. Moreover, each vertex is in at most r maximal bicliques; therefore, $v.bin, v.bicl_i.prev.index, v.bicl_i.nx.index \leq r$, and the label of v uses $O(\log L_1 + \log r + r \log L_2 + r \log r + r \log L_1) \in O(r \log L_1 + r \log L_2)$ bits. If L_1 and L_2 are polynomial in n, which we stated as an assumption in Section 3.1.3, then the label size of v reduces to $O(r \log n)$. In turn, the graph is represented using $O(rn \log n)$ bits.

Observe that a 1-bic is a complete bipartite graph. Given that the number of labelled and unlabelled complete bipartite graphs on n vertices are 2^{n-1} and $\lfloor \frac{n+1}{2} \rfloor$, respectively, our dynamic scheme is not space optimal for $r = 1$. As such, we implicitly assume that $r \geq 2$.

Using an argument similar to that found in Section 4.2.1, we can show that, for $r \geq 2$, there are $2^{\Omega(n \log n)}$ labelled r-bics on n vertices, thereby, our dynamic scheme for r-bics is space-optimal when $r \in O(1)$. Consider a graph consisting of a complete bipartite subgraph on $n - 1$ vertices and one vertex v adjacent to each of the $n - 1$ vertices in the complete bipartite subgraph. There are $n2^{n-2} \in 2^{\Omega(n \log n)}$ such graphs, each of which is a 2-bic. Yet, for $r' > r$, an r-bic is also an r'-bic; thereby, there are $2^{\Omega(n \log n)}$ labelled r-bics on n vertices.

For $r \in \omega(1)$, we cannot offer comment on optimality as we have no additional lower bounds. We also cannot offer comment on the optimality in the unlabelled case, when $r \in O(1)$, as the $2^{\Omega(n \log n)}$ lower bound has not yet been established on the number of unlabelled r-bics on n vertices.

5.2.2 Relabeller

Let us now examine the relabellers included in our dynamic scheme. As with the dynamic schemes presented for line graphs and r-minoes, detailed pseudocode appears in Appendix C. In the following discussion, G is the original graph and G' is the changed graph.

Deleting a vertex from the graph

Consider the following lemma, which describes how the deletion of a vertex affects the maximal bicliques in the graph.

Lemma 5.10 *Consider a graph G' formed by deleting a vertex v from a graph G, where X denotes the neighbourhood of v in G. Let $\mathcal{J}$ be the set of bicliques $\{B|B$ is a maximal biclique of G and $v \notin B\}$. The set of maximal bicliques of G' can be partitioned as $\mathcal{J} \cup \{B \setminus \{v\}|B$ is a maximal biclique of G, $v \in B$, and $B \setminus \{v\} \not\subseteq J$, for all $J \in \mathcal{J}\}$.*

Lemma 5.10, whose proof is identical to that of Lemma 5.1, suggests the design of our relabeller, DELETEVERTEX, as presented in Figure 5.7. As expected, DELETEVERTEX is almost identical to its sister algorithm presented in Section 5.2.

Despite their similarities, these two sister algorithms have some differences. Observe that every vertex in $V_{G'}$ belongs to some maximal biclique containing v. Therefore, we are able to determine V_G while determining if the only maximal biclique of G containing $B \setminus \{v\}$ is B, for each biclique $B \in \mathcal{B}$. Consequently, if $r \in \omega(1)$, then we can ensure that DELETEVERTEX is error-detecting by checking all vertex labels to confirm that each vertex belongs to at most r maximal cliques. This requires $\Theta(n)$ time, where $n \leq \sum_{B \ni v} |B|$.

In the case of r-minoes, the size of each maximal clique containing v was bounded by $|X|$, thereby leading to a running time of $O(v.cin \cdot \sum_{x \in X} x.cin) \in (O(r^2|X|)$. In the case of r-bics, the size of each maximal biclique containing v is bounded only by $\mathbf{B}$, the size of the largest maximal biclique in G. As such, for r-bics, DELETEVERTEX runs in $O(v.bin \cdot \underset{B \ni v}{max}\{\sum_{b \in B} b.bin\}) \in O(r^2\mathbf{B})$ time. Observe that DELETEVERTEX accesses $\Theta(\underset{B \ni v}{max}\{|B|\})$ labels, which require $\Omega(\sum_{B \ni v, b \in B} b.bin)$ bits in total. Therefore, the running time of DELETEVERTEX is polynomial in the size of its inputs.

Furthermore, in the case of r-bics, the modification locality of DELETEVERTEX is unbounded, as it is possible that some maximal bicliques of G containing v may have been independent sets.

```
DELETEVERTEX($G, v$)
Input: An adjacency labelling of an $r$-bic $G$ (that is, the labels thereof) created using our dynamic scheme, and a vertex $v$ in $V_G$. Note that the labels of $G$ are only accessed as required.
Output: An adjacency labeling of a graph $G'$ (again, the labels thereof) formed by deleting $v$ from $G$, providing $G'$ is an $r$-bic. If $G'$ is not an $r$-bic, then the output indicates as such.

$\mathcal{B} \leftarrow \{B | B$ is a maximal biclique containing $v\}$
for $B \in \mathcal{B}$ do
    if the only maximal biclique of $G$ containing $B \setminus \{v\}$ is $B$ then
        remove $v$ from $B$
    else eliminate $B$
delete $v$ and free its identifier
if $r \in \omega(1)$ then
    if some vertex belongs to more than $r$ maximal bicliques then
        error no longer an $r$-mino
```

Figure 5.7: The relabeller DELETEVERTEX which relabels an r-bic when a vertex is deleted

Adding a vertex.

Consider a vertex v which is to be added to a graph G, where X denotes the neighbourhood of v. For each biclique B of G, let $\{P_0^B, P_1^B\}$ be the partition of B defined by $b \in P_i^B$ if and only if $b \in X$ and $b \notin B_i$, or $b \notin X$ and $b \in B_i$. By adding v to the subset of B_i belonging to P_i^B, we obtain a biclique of the new graph. The maximality of such bicliques is addressed by the following lemma, which describes how the addition of a vertex affects the maximal bicliques in the graph.

Lemma 5.11 *Consider a graph G' formed by adding a vertex v to a graph G, where X denotes the neighbourhood of v. Let $\mathcal{I}$ be the set of bicliques $\{P_0^B, P_1^B | B$ is a maximal biclique of $G\}$. The set of maximal bicliques of G' can be partitioned as $\{I \cup \{v\} | I$ is a maximal element of $\mathcal{I}\} \cup \{B | B$ is a maximal biclique of G and $P_0^B, P_1^B \neq B\}$.*

Specifically, the maximal bicliques in G' of the form $P_i^B \cup \{v\}$, where $P_i^B = B$, are achieved by adding v to P_i^B. On the other hand, those maximal bicliques of the form $P_i^B \cup \{v\}$, where $P_i^B \neq B$, are achieved by creating an entirely new maximal biclique. By creating new maximal bicliques in these situations, we carry forward all those maximal bicliques of the form B, where B is a maximal biclique of G and $P_0^B, P_1^B \neq B$.

Proof. Consider a set of the form $I \cup \{v\}$, where I is a maximal element of $\mathcal{I}$, and consider a set of the form B, where B is a maximal biclique of G and $P_0^B, P_1^B \neq B$. By definition, $v \in I \cup \{v\}$, however, v is not a vertex of G. Therefore, $v \notin B$. Consequently, the two sets of the claimed partition are disjoint.

Since I is a biclique of G, where, without loss of generality, $I = P_0^I$, we know that $I \cup \{v\}$ is a biclique of G'. If $I \cup \{v\}$ is not a maximal biclique of G', then there exists some biclique

B^* of G' for which $I \cup \{v\} \subset B^*$. Since $v \in B^*$, we know that $I \subset B^* \setminus \{v\}$ and, without loss of generality, $B^* \setminus \{v\} = P_0^{B^*}$, which contradicts the maximality of I in $\mathcal{I}$. Therefore, $I \cup \{v\}$ is a maximal biclique of G'.

Since B is a biclique in G, it is also a biclique in G'. If B is not a maximal biclique of G', then there exists some biclique B^* of G' for which $B \subset B^*$. But $P_0^B, P_1^B \neq B$, so $P_0^{B^*}, P_1^{B^*} \neq B^*$, which gives $v \notin B^*$, as B^* is a biclique. Since $v \notin B^*$, B^* is also a biclique in G, which contradicts the maximality of B in G. Therefore, B is a maximal biclique of G'.

Having shown that the described sets are disjoint, and that their members are maximal bicliques of G', it remains to show that these sets contain all the maximal bicliques of G'.

Consider a maximal biclique B' of G'. If $v \notin B'$, then B' is maximal in G. Moreover, $P_0^B, P_1^B \neq B'$, otherwise, the existence of the biclique $B' \cup \{v\}$ in G' contradicts the maximality of B' in G'.

On the other hand, if $v \in B'$, consider the biclique $I' = B' \setminus \{v\}$ of G. Since $I' \in \{P_0^{I'}, P_1^{I'}\}$, there must be some maximal element I^*, of $\mathcal{I}$, that contains I'. If $I' \subset I^*$, then $B' = I' \cup \{v\} \subset I^* \cup \{v\}$, contradicting the maximality of B' in G'. Therefore, I' is a maximal element of $\mathcal{I}$, as desired. □

Lemma 5.11, and its proof, suggest the design of our relabeller, AddVertex, as presented in Figure 5.8. Perhaps the greater challenge in designing AddVertex lies in how to identify the maximal bicliques of G that do not contain any vertices of X, as we will need to determine these bicliques via the vertices of X.

To improve the running time of our algorithm, we insist that AddVertex represents X, the set of vertices to which v is made adjacent, in the same manner that S_1 was represented in our discussion of the reciprocal pointer technique found in Section 5.1.1. This one-time effort requires $\Theta(|X|)$ time, but will pay dividends later on.

The relabeller, AddVertex, first establishes the new vertex v. It does this in $O(1)$ time by assigning an identifier to v and setting $v.bin$ to 0. If $G = \emptyset$, then $V_{G'} = \{v\}$, therefore, the only new maximal biclique in G' is $\{v\}$. How AddVertex creates new maximal bicliques is discussed shortly.

Providing $G \neq \emptyset$, AddVertex first obtains $\mathcal{B} = \cup_{u \in V_G} \mathcal{B}_u$, the set of maximal bicliques of G. Let x^* be a member of X. Since every vertex in V_G belongs to some maximal biclique containing x^*, we can traverse each of the $x^*.bin$ bicliques of x^* to determine V_G. Where $\mathbf{B}$ is the size of the largest biclique in G, determining V_G in this manner takes $O(x^*.bin \cdot \mathbf{B}) \in O(r\mathbf{B})$ time.

Each $\mathcal{B}_u$ can be determined in $\Theta(u.bin) \in O(r)$ time, where $|\mathcal{B}_u| \leq r$. Therefore, $\mathcal{B}$ contains no more than $\sum_{u \in V_G} u.bin \leq rn$ entries, and can be determined in $O(\sum_{u \in V_G} u.bin) \in O(rn)$ time. Recall that, to efficiently extract information about a maximal biclique from

AddVertex(G, X)

Input: An adjacency labelling of an r-bic G (that is, the labels thereof) created using our dynamic scheme and a subset X of V_G. Note that the labels of G are accessed only as required.

Output: Let G' be the graph formed by adding a new vertex v to G, where v is adjacent to exactly those vertices in X. Providing G' is an r-bic, the output is an adjacency labelling of G' (again, the labels thereof). If G' is not an r-bic, the output indicates as such.

```
create a new vertex $v$
if $G = \emptyset$ then
    make a new maximal biclique $\{v\}$
else $\mathcal{B} \leftarrow \{B | B$ is a maximal biclique of $G\}$
    $\mathcal{D} \leftarrow \emptyset$
    for $B \in \mathcal{B}$ do
        $\mathcal{B} \leftarrow \mathcal{B} \setminus \{B\}$
        for $i \in \{0,1\}$ do
            if $P_i^B$ is not contained in any of the maximal bicliques in $\mathcal{B}$ or $\mathcal{D}$ then
                if $P_i^B \doteq B$ then
                    add $v$ to $B$
                else make a new maximal biclique $P_i^B \cup \{v\}$
        if either $P_0^B$ or $P_1^B$ was not contained in any of the maximal bicliques in $\mathcal{B}$ or $\mathcal{D}$ then
            $\mathcal{D} \leftarrow \mathcal{D} \cup \{B\}$
if $r \in o(1)$ then
    if some vertex belongs to more than $r$ maximal cliques then
```

Figure 5.8: The relabeller AddVertex which relabels an r-bic when a vertex is added

the circular doubly linked list of its vertices, we must know the identifier of some vertex that belongs to the maximal biclique, as well as the index of the corresponding *.bicl* entry. Consequently, for each entry $u.bicl_i.num$ in $\mathcal{B}$, we assume that we also retain a reference to u and i.

Just as we insisted that X be represented with reciprocal pointers, we also insist that $\mathcal{B}$ be represented with reciprocal pointers.The only difference between these reciprocal pointer representations is that the representation for $\mathcal{B}$ will need to be maintained dynamically. As a biclique B is chosen from $\mathcal{B}$ we remove it from $\mathcal{B}$; this removal from the reciprocal pointer representation takes O(1) time.

For each maximal biclique B in $\mathcal{B}$, we entertain the possibility of P_0^B and P_1^B being maximal elements of the set $\mathcal{I}$, seen in Lemma 5.11. For each element b in B, the membership of b in X and the value of $b.bicl_j.part$, where $b.bicl_j.num = B$, determine whether b belongs to P_0^B or P_1^B. Given the representation of X with reciprocal pointers, determining the membership of b in X requires O(1) time; moreover, the index j for which $b.bicl_j.num = B$ is known from the previous vertex in the circular doubly linked list about B. Therefore, traversing B to determine P_0^B and P_1^B requires $O(|B|)$ time.

If P_i^B is contained in some maximal biclique B^* remaining in $\mathcal{B}$, then we do nothing as P_i^B will present itself later. Specifically, if $P_i^B \subset P_j^{B^*}$, then P_i^B is not a maximal element of $\mathcal{I}$, and if $P_i^B = P_j^{B^*}$, then we avoid possible duplication of maximal bicliques. Similarly, if P_i^B is contained in some maximal clique B^* in $\mathcal{D}$, the set of bicliques D for which $D \cup \{v\}$ has been made a maximal biclique of G', then $P_i^B \subset B^*$ (given that B^* is selected from $\mathcal{B}$ before B, the addition of B^* to $\mathcal{D}$ was contingent on $B^* \not\subseteq P_i^B$).

The biclique P_i^B is contained in some maximal biclique in $\mathcal{B}$ or $\mathcal{D}$ if and only if $\cap_{u \in P_i^B}(\mathcal{B}_u \cap (\mathcal{B} \cup \mathcal{D})) \neq \emptyset$. Since $|\mathcal{B}_u| = u.bin$, and $\mathcal{B}$ and $\mathcal{D}$ are represented using reciprocal pointers, each set $\mathcal{B}_u \cap (\mathcal{B} \cup \mathcal{D})$ can be determined in $\Theta(u.bin) \in O(r)$ time. In turn, the condition $\cap_{u \in P_i^B}(\mathcal{B}_u \cap (\mathcal{B} \cap \mathcal{D})) \neq \emptyset$ can be checked in $O(u.bin \cdot |P_i^B|) \in O(r|P_i^B|) \in O(r|B|)$ time using the reciprocal pointer technique discussed in Section 5.1.1.

Now if P_i^B is not contained in any maximal bicliques of $\mathcal{B} \cup \mathcal{D}$, then we make a maximal biclique of $P_i^B \cup \{v\}$. Exactly how we make a maximal biclique of $P_i^B \cup \{v\}$ depends on whether $P_i^B = B$; note that this condition can be checked while computing P_i^B. If $P_i^B = B$, then we simply add v to B, as B will no longer be maximal in G'. Specifically, AddVertex inserts v into the circular doubly linked list for B such that the *.part* value of the corresponding *v.bicl* entry is i, then increases the value of *v.bin* by one. This can be done in O(1) time. Otherwise, if $P_i^B \neq B$, then we form a new maximal biclique, $P_i^B \cup \{v\}$, as B will continue to be maximal in G'. Specifically, AddVertex increases the value of the *.bin* counter of each of the vertices in $P_i^B \cup \{v\}$ by one, establishes a circular doubly linked list of the vertices in $P_i^B \cup \{v\}$, and creates a new *.bicl* entry for each member of

$P_i^B \cup \{v\}$ (mirroring the *.part* values seen in B, and setting the appropriate *.part* value of v to i) while establishing the circular doubly linked list. Forming this new maximal biclique takes $O(|P_i^B|) \in O(|B|)$ time.

Whenever a *.bin* counter is increased, we check that its value is no greater than r, thereby, ADDVERTEX is error-detecting. Moreover, if $r \in o(1)$, then our previous determination of V_G can be used to check all labels to ensure that each vertex belongs to at most r maximal cliques. The time required to do this is $\Theta(n)$.

The total running time of ADDVERTEX is $O(x^*.bin \cdot \mathbf{B} \cdot \sum_{u \in V_G} u.bin) \in O(r^2 n \mathbf{B}) \in O(r^2 n^2)$. Given that ADDVERTEX accesses the entire labelling, its running time is polynomial in the size of its input.

From Corollary 3.8, we see that ADDVERTEX gives polynomial time recognition for r-bics.

Theorem 5.12 *For any graph G on n vertices, we can determine if G is an r-bic in $O(r^2 n^3)$ time.*

Furthermore, observe that we can use ADDVERTEX, just as we did in the proof of Theorem 3.6, to create a $O(r^2 n^3)$ time marker for our dynamic scheme.

Deleting an edge.

Consider the following lemma, which describes how the deletion of an edge affects the maximal bicliques in the graph.

Lemma 5.13 *Consider a graph G' formed by deleting an edge uv from a graph G, where $V_G \neq \{u, v\}$. Let X_u and X_v denote the neighbourhoods of u and v in G', respectively, and let W denote the vertices of G for which $w \in W$ if and only if $w \in X_u \Leftrightarrow w \in X_v$. Furthermore, let $\mathcal{L}_1$ be $\{B | B$ is a maximal biclique of G, $B \not\subseteq W$, and $|\{u, v\} \cap B| = 1\}$, let $\mathcal{L}_2$ be $\{B | B$ is a maximal biclique of G and $u, v \notin B\}$, and let $\mathcal{K}$ be the set of bicliques $\{(B \setminus \{v\}) \cap W | B$ is a maximal biclique of G, $u \in B\}$.*

The set of maximal bicliques of G' can be partitioned as $\mathcal{L}_1 \cup \mathcal{L}_2 \cup \{B \setminus \{u\} | B$ is a maximal biclique of G, $u, v \in B$, $B \neq \{u, v\}$, and $B \setminus \{u\} \not\subset B_1$, for all maximal bicliques B_1 of G, $B_1 \neq B\} \cup \{B \setminus \{v\} | B$ is a maximal biclique of G, $u, v \in B$, $B \neq \{u, v\}$, and $B \setminus \{v\} \not\subset B_1$, for all maximal bicliques B_1 of G, $B_1 \neq B\} \cup \{K \cup \{v\} | K$ is a maximal element of $\mathcal{K}\}$.

Unlike Lemma 5.1 which tells us that maximal cliques can only be destroyed when an vertex is deleted, Lemma 5.10 indicates that the deletion of an edge can cause bicliques to be both created and destroyed. As such, DELETEEDGE has the flavour of both ADDVERTEX, an algorithm in which maximal bicliques get created, and DELETEVERTEX, an algorithm in which maximal bicliques get destroyed.

In Lemma 5.10, $\mathcal{L}_1$ and $\mathcal{L}_2$ are the sets of maximal bicliques that are unaltered.

For any maximal biclique B of G containing u and v, where $B \neq \{u,v\}$, we need to consider whether $B \setminus \{u\}$ and $B \setminus \{v\}$ will continue to be maximal in G'. Specifically, if they are properly contained in another maximal biclique besides B, then they will no longer be maximal in G', and must be destroyed.

For any maximal biclique of the form $((B \setminus \{v\}) \cap W) \cup \{v\} = (B \cap W) \cup \{v\}$, where $u \in B$, we consider four cases. If $B \subseteq W$ and $v \in B$, then $B = \{u,v\}$, so we merely switch the value of $v.part$ that corresponds to B. If $B \subseteq W$ and $v \notin B$, then $(B \cap W) \cup \{v\}$ will be created by adding v to B such that u and v belong to different parts of B. If $B \nsubseteq W$ and $v \in B$, then B no longer remains maximal, so we change B into $(B \cap W) \cup \{v\} = \{u,v\}$. Finally, if $B \nsubseteq W$ and $v \notin B$, then we create a new maximal biclique $(B \cap W) \cup \{v\}$, as B will continue to remain maximal in G'.

Proof. Consider the following items.

- An element L_1 of $\mathcal{L}_1$.
- An element L_2 of $\mathcal{L}_2$.
- A set of the form $K \cup \{v\}$, where K is a maximal element of $\mathcal{K}$.
- A set $B_u \setminus \{u\}$, where B_u is a maximal biclique of G, $u, v \in B_u$, $B_u \neq \{u,v\}$, and $B_u \setminus \{u\} \not\subset B_1$, for all maximal bicliques B_1 of G for which $B_1 \neq B_u$.
- A set $B_v \setminus \{v\}$, where B_v is a maximal biclique of G, $u, v \in B_v$, $B_v \neq \{u,v\}$, and $B_v \setminus \{v\} \not\subset B_1$, for all maximal bicliques B_1 of G for which $B_1 \neq B_v$.

First, observe that $u \in W$. Therefore, $u, v \in K \cup \{v\}$, which is also to say that $|\{u,v\} \cap (K \cup \{v\})| = 2$. By definition, $|\{u,v\} \cap L_1| = 1$ and $|\{u,v\} \cap L_2| = 0$, therefore, L_1, L_2, and $K \cup \{v\}$ are pairwise unequal. As well, $v \in B_u \setminus \{u\}$ and $u \in B_v \setminus \{v\}$; therefore, $B_u \setminus \{u\}$, $B_v \setminus \{v\}$, and $K \cup \{v\}$ are pairwise unequal. Moreover, both L_1 and L_2 are maximal bicliques of G, whereas $B_u \setminus \{v\}$ and $B_v \setminus \{v\}$ are not, as they are contained in bicliques B_u and B_v, respectively. Therefore, L_1, L_2, $B_u \setminus \{v\}$, and $B_v \setminus \{v\}$ are pairwise unequal. Consequently, the sets of the claimed partition are disjoint.

Since K is a biclique of G, where $K \subseteq W$, $K \cup \{v\}$ is a biclique of G'. If $K \cup \{v\}$ is not a maximal biclique of G', then there exists some biclique B^* of G' for which $K \cup \{v\} \subset B^*$. Since $u, v \in K \cup \{v\}$, we know that $u, v \in B^*$; therefore, $B^* \setminus \{v\} \subseteq W$, where $B^* \setminus \{v\}$ is a biclique of G. But $K \subset B^* \setminus \{v\}$, which contradicts the maximality of K in $\mathcal{K}$. Therefore, $K \cup \{v\}$ is a maximal biclique of G'.

Since L_1 is a biclique in G and $|\{u,v\} \cap L_1| = 1$, it is also a biclique in G'. If L_1 is not a maximal biclique of G', then there exists some biclique B^* of G' for which $L_1 \subset B^*$. Since $L_1 \nsubseteq W$, we know that $B^* \nsubseteq W$; therefore, at least one of u and v does not belong to B^*. This gives that B^* is a biclique of G, which contradicts the maximality of L_1 in G. Therefore, L_1 is a maximal biclique of G'.

Since L_2 is a biclique in G and $|\{u,v\} \cap L_2| = 0$, it is also a biclique in G'. If L_2 is not a maximal biclique of G', then there exists some biclique B^* of G' for which $L_2 \subset B^*$. If at least one of u and v does not belong to B^*, then B^* is biclique of G, which contradicts the maximality of L_2 in G. On the other hand, if both u and v belong to B^*, then $L_2 \subset B^* \setminus \{u\}$, as $u, v \notin L_2$. But $B^* \setminus \{u\}$ is a biclique of G, which contradicts the maximality of L_2 in G. Therefore, L_2 is a maximal biclique of G'.

Since B_u is a biclique in G, so too is $B_u \setminus \{u\}$ in G'. If $B_u \setminus \{u\}$ is not a maximal biclique of G', then there exists some biclique B^* of G' for which $B_u \setminus \{u\} \subset B^*$. Given that $u, v \in B_u$, we know that $B_u \cap W = \{u, v\}$ as the edge uv belongs to G. However, by definition, $B_u \neq \{u, v\}$; therefore, B_u must contain some vertex y which does not belong to W. Since $B_u \setminus \{u\} \subset B^*$, B^* also contains y and v, which means that B^* cannot contain u. Therefore, B^* is a biclique in G, where $B^* \neq B_u$, yet this contradicts the definition of $B_u \setminus \{u\}$. Therefore, $B_u \setminus \{u\}$ is a maximal biclique of G'. A similar argument gives that $B_v \setminus \{v\}$ is a maximal biclique of G'.

Having shown that the described sets are disjoint, and that their members are maximal bicliques of G', it remains to show that these sets contain all the maximal bicliques of G'.

Consider a maximal clique B' of G'. If $u, v \notin B'$, then B' is maximal in G, where $B' \in \mathcal{L}_2$.

If $u \in B'$, $v \notin B'$, and $B' \not\subseteq W$, then v cannot be added to B' to get a larger biclique in G. That is, B' is maximal in G, where $B' \in \mathcal{L}_1$. Similarly, if $u \notin B'$, $v \in B'$, and $B' \not\subseteq W$, then B' is maximal in G, where $B' \in \mathcal{L}_1$.

If $u \in B'$, $v \notin B'$, and $B' \subseteq W$, then $B' \cup \{v\}$ is a maximal biclique of G. Moreover, $B \neq \{u, v\}$, as $B' \neq \{u\}$ (a vertex cannot be in a biclique by itself). Similarly, if $u \notin B'$, $v \in B'$, and $B' \subseteq W$, then $B' \cup \{u\}$ is a maximal biclique of G, where $B \neq \{u, v\}$.

If $u, v \in B'$, then $B' \setminus \{v\}$ is a biclique of G, where $B' \setminus \{v\} \subseteq W$ and $B' \setminus \{v\}$ is contained in some maximal biclique B^* of G. Now $B' \setminus \{v\} \subseteq B^* \setminus \{v\}$ and $B' \setminus \{v\} \subseteq W$, so $B' \setminus \{v\} \subseteq (B^* \setminus \{v\}) \cap W$. If $B' \setminus \{v\} \subset (B^* \setminus \{v\}) \cap W$, then there exists some element w of B^* for which $w \in W$ and $w \in B^* \setminus \{v\}$ (therefore, $w \neq v$), but $w \notin B' \setminus \{v\}$. However, the existence of the biclique $B' \cup \{w\}$ in G' contradicts the maximality of B'. Therefore, $B' \setminus \{v\} = (B^* \setminus \{v\}) \cap W$.

Moreover, if there exist some K^* in $\mathcal{K}$, such that $B' \setminus \{v\} \subset K^*$, then there exists some element w of K^* (therefore, $w \in W$, $w \neq v$), but $w \notin B' \setminus \{v\}$. Again, the existence of the biclique $B' \cup \{w\}$ in G' contradicts the maximality of B'. Therefore, $B' = K \cup \{v\}$, where K is a maximal element of $\mathcal{K}$, as desired. □

Lemma 5.10, and its proof, suggest the design of our relabeller, DELETEEDGE, as presented in Figure 5.9.

DELETEEDGE(G, u, v)

Input: An adjacency labelling of an r-bic G (that is, the labels thereof) created using our dynamic scheme, and two distinct vertices u and v of V_G for which $uv \in E_G$. Note that the labels of G are only accessed as required.

Output: An adjacency labeling of a graph G' (again, the labels thereof) formed by deleting the edge uv from G, providing G' is an r-bic. If G' is not an r-bic, then the output indicates as such.

```
$\mathcal{B} \leftarrow \{B | B$ is a maximal biclique containing both $u$ and $v$ and $B \neq \{u, v\}\}$
$\mathcal{C} \leftarrow \{B | B$ is a maximal biclique containing $u\}$
for $B \in \mathcal{B}$ do
    if the only maximal biclique of $G$ containing $B \setminus \{v\}$ is $B$ then
        remove $v$ from $B$
        if the only maximal biclique of $G$ containing $B \setminus \{u\}$ is $B$ then
            make a new maximal biclique of $B \setminus \{u\}$
    elseif the only maximal biclique of $G$ containing $B \setminus \{u\}$ is $B$ then
        remove $u$ from $B$
    else eliminate $B$
$X_u \leftarrow \{x | x$ is a neighbour of $u$ (in $G'$)$\}$
$X_v \leftarrow \{x | x$ is a neighbour of $v$ (in $G'$)$\}$
$W \leftarrow \{w | w \in V_G$ and $w \in X_u \Leftrightarrow w \in X_v\}$
$\mathcal{D} \leftarrow \emptyset$
for $B \in \mathcal{C}$ do
    $\mathcal{C} \leftarrow \mathcal{C} \setminus \{B\}$
    if $(B \setminus \{v\}) \cap W$ is not contained in any of the maximal bicliques in $\mathcal{C}$ or $\mathcal{D}$ then
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{B\}$
        if $B \subseteq W$ then
            if $v \in B$ then
                switch the maximal biclique $B = \{\{u\}, \{v\}\}$ to $\{\{u, v\}, \emptyset\}$
            else add $v$ to $B$ such that $v$ belongs to the same part as $u$
        elseif $v \in B$ then
            $B \leftarrow \{\{u, v\}, \emptyset\}$
        else create a new maximal biclique $(B \cap W) \cup \{v\}$
```

Figure 5.9: The relabeller DELETEEDGE which relabels an r-bic when an edge is deleted

The relabeller, DELETEEDGE, first obtains $\mathcal{B} = \mathcal{B}_u \cap \mathcal{B}_v$, the set of maximal bicliques containing both u and v. Since $\mathcal{B}_u$ and $\mathcal{B}_v$ can be determined in $\Theta(u.bin) \in O(r)$ and $\Theta(v.bin) \in O(r)$ time, respectively, where $|\mathcal{B}_u| = u.bin \leq r$ and $|\mathcal{B}_v| = v.bin \leq r$, $\mathcal{B}$ can be determined in $O(u.bin + v.bin) \in O(r)$ time using the reciprocal pointer technique discussed in Section 5.1.1.

As well, DELETEEDGE, also obtains $\mathcal{C} = \mathcal{B}_u$, the set of maximal bicliques containing u, which can be determined while obtaining $\mathcal{B}$. We insist that $\mathcal{C}$ be represented using dynamic reciprocal pointers, which take $\Theta(u.bin) \in O(r)$ time to establish.

For each maximal clique B of $\mathcal{B}$, providing $B \neq \{u, v\}$, the bicliques $B \setminus \{u\}$ and $B \setminus \{v\}$ are examined to determine if they maximal in G'. For $\alpha \in \{u, v\}$, $B \setminus \{\alpha\}$ can be determined in $O(|B|)$ time by traversing B; moreover, while determining $B \setminus \{\alpha\}$, we can confirm that $B \neq \{u, v\}$. The biclique $B \setminus \{\alpha\}$ is maximal in G' if and only if the set $\mathcal{A}_\alpha = \bigcap_{b' \in B \setminus \{\alpha\}} (\mathcal{B}_{b'} \setminus B)$ is empty. As per our discussion in Section 5.1.1, $\mathcal{A}_\alpha$ can be determined in $O(\sum_{b' \in B \setminus \{\alpha\}} b'.bin) \in O(r|B \setminus \{\alpha\}|) \in O(r|B|)$ time.

First, let us assume that exactly one of $B \setminus \{u\}$ and $B \setminus \{v\}$ is a maximal biclique in G'; without loss of generality, let the maximal biclique be $B \setminus \{v\}$. We develop this maximal biclique by removing v from B, just as we did in DELETEVERTEX, using $O(1)$ time. As a second possibility, consider if neither $B \setminus \{u\}$ nor $B \setminus \{v\}$ is a maximal clique in G'. In this case, we eliminate the maximal clique B, just as we did in DELETEVERTEX, using $O(|B|)$ time. Finally, if $B \setminus \{u\}$ and $B \setminus \{v\}$ are both maximal in G', then we develop $B \setminus \{v\}$ by removing v from C, using $O(1)$ time; however, $B \setminus \{u\}$ must be developed by establishing a new maximal biclique, just as we did in ADDVERTEX, requiring $O(|B'_u|) \in O(|B|)$ time. Whenever a *.bin* counter is increased during the creation of the new maximal biclique, we check that its value is no greater than r, thereby, the efforts of DELETEEDGE on $\mathcal{B}$ are error-detecting. Since $\mathcal{B}$ contains at most $\min\{u.bin, v.bin\}$ maximal bicliques, the running time of DELETEEDGE on $\mathcal{B}$ is $O(\min\{u.bin, v.bin\} \cdot \max_{B \ni u,v} \{\sum_{b \in B} b.bin\}) \in O(r^2\mathbf{B})$.

For each maximal clique B of $\mathcal{C}$, we entertain the possibility of $B' = (B \setminus \{v\}) \cap W$ being a maximal element of the set $\mathcal{K}$, seen in Lemma 5.13. The biclique B' can be computed in $O(|B|)$ time by traversing B and testing adjacency with u and v to determine membership in W.

If B' is contained in some maximal biclique B^* remaining in $\mathcal{C}$ (observe that DELETEEDGE removes B from $\mathcal{C}$ when it is selected), then we do nothing as B' will present itself later. Specifically, if $B' \subset B^{*\prime}$, then B' is not a maximal element of $\mathcal{K}$; otherwise, if $B' = B^{*\prime}$, then we avoid possible duplication of maximal bicliques. Similarly, if B' is contained in some maximal clique B^* in $\mathcal{D}$, the set of bicliques D for which $D' \cup \{v\}$ has been made a maximal biclique of G', then $B' \subset B^{*\prime}$ (given that B^* is selected from $\mathcal{C}$ before B, the addition of B^* to $\mathcal{D}$ was contingent on $B^{*\prime} \not\subseteq B'$). Just as $\mathcal{C}$ is represented using dynamic reciprocal pointers, we insist that $\mathcal{D}$ also be represented using dynamic reciprocal pointers.

The biclique B' is contained in some maximal biclique in $\mathcal{C}$ or $\mathcal{D}$ if and only if $\cap_{b' \in B'} (\mathcal{B}_{b'} \cap (\mathcal{C} \cup \mathcal{D})) \neq \emptyset$. Since $|\mathcal{B}_{b'}| = b'.bin$, and $\mathcal{C}$ and $\mathcal{D}$ are represented using reciprocal pointers, each set $\mathcal{B}_{b'} \setminus (\mathcal{C} \cup \mathcal{D})$ can be determined in $\Theta(b'.bin) \in O(r)$ time. In turn, the condition $\cap_{b' \in B'} (\mathcal{B}_{b'} \cap (\mathcal{C} \cap \mathcal{D})) \neq \emptyset$ can be checked in $O(\sum_{b' \in B'} b'.bin) \in O(r|B'|) \in O(r|B|)$ time using the reciprocal pointer technique discussed in Section 5.1.1.

Now if B' is not contained in any maximal biclique remaining in $\mathcal{B}$, then we make a maximal biclique of $B' \cup \{v\}$. Exactly how we make a maximal biclique of $B' \cup \{v\}$ depends on whether $B \subseteq W$ and $v \in B$. These conditions can be checked while computing B'.

If $B \subseteq W$ and $v \in B$, then $B = \{u, v\}$, so we merely switch the value of *v.part* that corresponds to B. If $B \subseteq W$ and $v \notin B$, then we simply add v to B as B will no longer be maximal in G'. Specifically, DELETEEDGE inserts v into the circular doubly linked list for B, increases the value of *v.bin* by 1, and sets the value of $v.bicl_i.part$ to that of $u.bicl_j.part$ for the values of i and j that correspond to B. This can be done in O(1) time. If $B \not\subseteq W$ and $v \in B$, then B no longer remains maximal, so we change B into $(B \cap W) = \{u, v\}$. Specifically, DELETEEDGE removes all the vertices except u and v from the circular doubly linked list for B, while decreasing their *.bin* values by 1, then sets the value of $v.bicl_i.part$ to that of $u.bicl_j.part$ for the values of i and j that correspond to B. This can be done in $\mathrm{O}(|B|)$ time. Finally, if $B \not\subseteq W$ and $v \notin B$, then we create a new maximal biclique $(B \cap W) \cup \{v\}$, as B will continue to remain maximal in G'. Specifically, DELETE increases the value of the *.bin* counter of each of the vertices in $B' \cup \{v\}$ by one, establishes a circular doubly linked list of the vertices in $B' \cup \{v\}$, and creates a new *.bicl* entry identical to that found in B, for each member of $B' \cup \{v\}$ while establishing the circular doubly linked list (in the case of v it sets the value of $v.bicl_i.part$ to that of $u.bicl_j.part$ for the values of i and j that correspond to $B' \cup \{v\}$); forming this new maximal clique takes $\mathrm{O}(|B'|) \in \mathrm{O}(|B|)$ time.

Whenever a *.bin* counter is increased, we check that its value is no greater than r, thereby, the efforts of DELETEEDGE on $\mathcal{C}$ are error-detecting. Since $|\mathcal{C}| = u.bin \leq r$, the running time of DELETEEDGE on $\mathcal{C}$ is $\mathrm{O}(u.bin \cdot \underset{B \ni u}{max}\{\sum_{b \in B} b.bin\}) \in \mathrm{O}(r^2\mathbf{B})$. Therefore, the total running time of DELETEEDGE is $\mathrm{O}(u.bin \cdot \underset{B \ni u}{max}\{\sum_{b \in B} b.bin\} + v.bin \cdot \underset{B \ni u}{max}\{\sum_{b \in B} b.bin\}) \in \mathrm{O}(r^2\mathbf{B})$. Observe that DELETEEDGE accesses $\Theta(\underset{B \ni u}{max}\{|B|\})$ vertex labels which require $\Omega(v.bin + \sum_{B \ni u, b \in B} b.bin\})$ bits in total. Thereby, the running time of DELETEEDGE is polynomial in the size of its inputs.

Adding an edge.

Consider the following lemma, which describes how the addition of an edge affects the maximal bicliques in the graph.

Lemma 5.14 *Consider a graph G' formed by adding an edge uv to a graph G, where $V_G \neq \{u, v\}$. Let X_u and X_v denote the neighbourhoods of u and v in G', respectively, and let W denote the vertices of G for which $w \in W$ if and only if $w \in X_u \Leftrightarrow w \notin X_v$. Furthermore, let $\mathcal{L}_1$ be $\{B | B$ is a maximal biclique of G, $B \not\subseteq W$, and $|\{u, v\} \cap B| = 1\}$, let $\mathcal{L}_2$ be $\{B | B$ is a maximal biclique of G and $u, v \notin B\}$, and let $\mathcal{K}$ be the set of bicliques $\{(B \setminus \{v\}) \cap W | B$ is a maximal biclique of G, $u \in B\}$.*

The set of maximal bicliques of G' *is* $\mathcal{L}_1 \cup \mathcal{L}_2 \cup \{B \setminus \{u\} | B$ *is a maximal biclique of* G, $u, v \in B$, $B \neq \{u, v\}$, *and* $B \setminus \{u\} \not\subset B_1$, *for all maximal bicliques* B_1 *of* G, $B_1 \neq B\} \cup \{B \setminus \{v\} | B$ *is a maximal biclique of* G, $u, v \in B$, $B \neq \{u, v\}$, *and* $B \setminus \{v\} \not\subset B_1$, *for all maximal bicliques* B_1 *of* G, $B_1 \neq B\} \cup \{K \cup \{v\} | K$ *is a maximal element of* $\mathcal{K}\}$.

The similarity between Lemmas 5.13 and 5.14, which differ only in their definition of the set W, suggest a relabeller AddEdge that is virtually identical to that of DeleteEdge. As such, AddEdge would also be error-detecting and have a running time of $O(r^2\mathbf{B})$.

5.3 Summary

In this chapter, we apply the circular doubly linked list technique seen in Chapter 4 to maximal cliques and maximal bicliques in order to develop error-detecting dynamic adjacency labelling schemes for r-minoes and r-bics, respectively. Both dynamic schemes use $O(r \log n)$ bit labels.

In the case of r-minoes, edge addition and deletion can be handled in $O(r^2\mathbf{D})$ time, vertex addition in $O(r^2e^2)$ time, and vertex deletion in $O(r^2e)$ time, where $\mathbf{D}$ is the maximum degree of the vertices in the original graph and e is the number of edges added to, or deleted from, the original graph. Unfortunately, if $r \in \omega(1)$, then our vertex deletion relabeller is not error-detecting. Similarly, if $r \in o(1)$, then our vertex addition relabeller is not error detecting.

In the case of r-bics, edge addition and deletion, as well as vertex deletion can be handled in $O(r^2\mathbf{B})$ time, and vertex addition in $O(r^2n\mathbf{B})$ time, where $\mathbf{B}$ is the size of the largest biclique in the original graph. Given these running times, one might be led to believe that vertex addition could just as easily be performed by adding an isolated vertex, then adding individual edges. However, by doing so the graph may escape the class of r-bics.

Chapter 6

Proper interval graphs

In this chapter, we develop a dynamic adjacency labelling scheme for proper interval graphs that allows the addition and deletion of vertices and edges. The labels used in this scheme require $O(\log n)$ bits, and updates require in $O(n)$ time. In comparison, the best known (static) adjacency labelling scheme for proper interval graphs is the scheme presented for interval graphs in Chapter 1 [45], which uses $O(\log n)$ bit labels and requires as much as $\Theta(n + m)$ time to generate a labelling (here we presume that the marker is input with only the proper interval graph, perhaps as an adjacency matrix, and must use a $\Theta(n + m)$ time algorithm like that of Corneil, Kim, Natarjan, Olariu, and Sprague [12] to determine the proper interval representation from the graph itself.) Proper interval graphs have been shown useful in the study of problems in genetics and psychology; a good starting point for information on the application of proper interval graphs is the text of McKee and McMorris [43].

A graph is a proper interval graph if it has an interval representation in which no interval contains another interval. Proper interval graphs can also be characterized using structures known as astral triples. An astral triple is a set of three vertices for which each pair are connected by a path in which no two consecutive vertices belong to the closed neighbourhood of the third vertex. In the graph shown in Figure 6.1, the bold vertices form an astral triple.

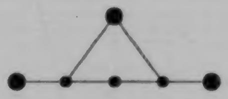


Figure 6.1: An astral triple. The bold vertices indicate the astral triple

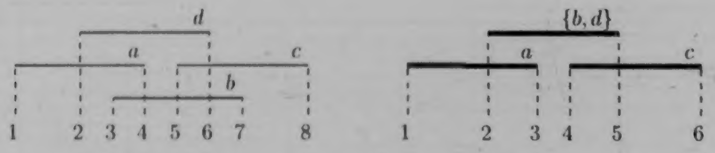
Perhaps the simplest example of an astral triple is $K_{1,3}$, often referred to as a claw. In the case of $K_{1,3}$, the three pendant vertices form the astral triple. The relationship between proper interval representations and astral triples is explicitly addressed in the following theorems.

Theorem 6.1 *[28] A graph is a proper interval graph if and only if it contains no astral triple.*

Theorem 6.2 *[49] An interval graph is proper if and only if it contains no induced* $K_{1,3}$.

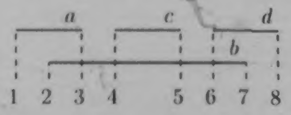
Another characterization of interval graphs is based on the notion of blocks [14]. For any graph G, consider the equivalence relation R, on V_G, defined by uRv if and only if $N[u] = N[v]$. This equivalence relation partitions the vertices into equivalence classes known as *blocks*. For example, the blocks of the proper interval graph represented in Figure 6.2(a) are $\{a\}$, $\{b, d\}$, and $\{c\}$. Ultimately, we can consider each block as a "mega-interval", as depicted in Figure 6.2(b).

Two blocks B and B' of a graph $G$are said to be *adjacent* if there exists an edge bb' of G for which b is in B and b' is in B' (consequently, if B is adjacent to B', then, for all b in B and all b' in B', b is adjacent to b'). In an extension of conventional graph terminology, we say that the (open) neighbourhood of a block is the set of blocks that are adjacent to it, and that its closed neighbourhood is its open neighbourhood unioned with itself. Similarly, we say that the degree of a block is the cardinality of its open neighbourhood, where $deg(B)$ denotes the degree of B.



(a) A proper interval representation.

(b) Blocks as "mega-intervals".



(c) An interval representation, which is not, and cannot be, proper.

Figure 6.2: Interval representations and blocks

As preliminary observations, consider the following properties of blocks.

Lemma 6.3 *The induced subgraph formed on the vertices of a block is a clique.*

Proof. Consider any two vertices u and v belonging to the same block of a graph. By definition, $N[u] = N[v]$, therefore, u and v are adjacent. The result follows. □

Lemma 6.4 *No component of a graph can be comprised of only two blocks.*

Proof. Consider a component C consisting of two blocks B_1 and B_2. If B_1 is not adjacent to B_2, then B_1 is a component itself, thereby, C is not a component. If B_1 is adjacent to B_2, then $B_1 \cup B_2$ forms a clique. Therefore, $B_1 = B_2$, which is also a contradiction. ☐

Lemma 6.5 *No two blocks can be adjacent to the same set of blocks.*

Proof. Consider two blocks B_1 and B_2, which are adjacent to the same set of blocks. For any vertices b_1 in B_1 and b_2 in B_2, $N[b_1] = N[b_2]$. Therefore, $B_1 = B_2$, which is a contradiction. ☐

A *straight enumeration* of a graph is a linear ordering of its blocks such that, for every block, the blocks in its closed neighbourhood are consecutive. In the case of the proper interval graph represented in Figure [illegible].2(a), the straight enumerations are $\Phi = \{a\} \prec \{b,d\} \prec \{c\}$ and $\Phi^R = \{c\} \prec \{b,d\} \prec \{a\}$, where Φ^R denotes the reversal of the straight enumeration Φ. The following theorem characterizes proper interval graphs in terms of straight enumerations.

Theorem 6.6 *[14] A graph is a proper interval graph if and only if it has a straight enumeration. Moreover, a connected proper interval graph has a unique straight enumeration (up to reversal).*

Hell, Shamir, and Sharan [27], on whose work we will heavily rely, refer to a straight enumeration of a connected proper interval graph as a *contig*.

Fundamental to our entire work on proper interval graphs is the following lemma, referred to as the "umbrella property". Given the frequency with which we use Lemma 6.7, we will only explicitly reference this lemma in the beginning of our discussion, or when the use of the lemma is not entirely obvious.

Lemma 6.7 *[41] Consider a straight enumeration Φ of a connected proper interval graph G. If B_1, B_2, and B_3 are blocks of G, such that $B_1 \prec B_2 \prec B_3$ in Φ and B_1 is adjacent to B_3, then B_2 is adjacent to B_1 and to B_3.*

6.1 Vertex labels, marker, and decoder

Our scheme closely resembles a dynamic representation of proper interval graphs due to Hell, Shamir, and Sharan [27] (their representation does not permit implicit adjacency testing from vertex labels). For each component of the proper interval graph, they maintain a data structure to represent a contig. In each contig, the first and last blocks are called *end blocks* and their members are *end vertices*; all other blocks are referred to as *inner blocks* and their members are *inner vertices*. Specifically, the data structure used by Hell et al. consists of the following.

- For each vertex, they maintain the name of its block.
- For each block, they maintain the following information.
    - The size of the block.
    - Left and right *near pointers* which point to the adjacent blocks immediately to the left and right, respectively, in the straight enumeration.
    - Left and right *far pointers* which point to the furthest adjacent blocks to the left and right, respectively, in the straight enumeration.
    - Left and right *self pointers* which point to the block itself.
    - An *end pointer* which is null if the block is an inner block of its contig, otherwise, it points to the other end block in the contig.

Unfortunately, we do not have the liberty of using pointers at the block level, rather, we must do so at the vertex level. To this effect, we select a *pointer vertex* $P(B)$ from each block B. If we wish to include a pointer Q from block B to block B', then we include that pointer in the label of $P(B)$, such that $Q(P(B)) = b'$, where $b' \in B'$. In essence, we create a "distributed" pointer.

Specifically, our labelling scheme is as follows.

- For each vertex v, we maintain the following.
    - A unique identifier for each vertex. Letting $\mathcal{L}$ be the number of bits required to represent the largest identifier, the uniqueness of the identifiers ensures that $\mathcal{L} \in \Omega(\log n)$. Given our assumption on the size of identifiers, as stated in Section 3.1.3, $\mathcal{L} \in \Theta(\log n)$.
    - The identifier of the block to which it belongs. Although we do not differentiate between a vertex and its identifier, we will differentiate between a block and its identifier, as the identifier of a block may change over time while we maintain a straight enumeration. To this effect, we denote the block containing v by $B(v)$, and denote the identifier of $B(v)$ by $b(v)$.

      Just as we required a unique identifier for each vertex, we require a unique identifier for each block. Where $\mathcal{B}$ is the size of the largest identifier, we ensure that $\mathcal{B} \in O(\log n)$.
    - The identifiers of the furthest adjacent blocks to the left and right of $B(v)$, denoted $f_L(v)$ and $f_R(v)$, respectively. This information requires $O(\mathcal{B})$ bits.

- For each block B, we encode the following information via the vertex labels.
  - The vertices in each block. This information is represented using a circular doubly linked list of the vertices in each block, mirroring the technique used previously in Chapters 4 and 5. This circular doubly linked list adds $O(\mathcal{L})$ bits to the label of each vertex. For each vertex v, we denote the next and previous vertices in the circular doubly linked list of $B(v)$ by $nx(v)$ and $prev(v)$, respectively. Again, we will say that we traverse B, although we really mean that we traverse the circular doubly linked list of vertices in B.
  - A pointer vertex, denoted $P(B)$. The label of the pointer vertex must contain a bit to denote that it is a pointer vertex. All other vertices in the block, contain the identifier of $P(B)$, as well as a bit to denote that they are not the pointer vertex of B. This information adds $O(1)$ bits to the label of the pointer vertex, and $O(\mathcal{L})$ bits to the label of all other vertices.

    To clarify how these distributed pointers are used at the vertex level, let us consider a pointer Q and a vertex v. The label of v will contain the identifier of $P(B(v))$, the pointer vertex of the block containing v (assuming $v \neq P(v)$); for simplicity, we will shorten $P(B(v))$ to $P(v)$. The label of $P(v)$ will contain the identifier of $Q(P(v))$, which we will similarly shorten to $Q(v)$. For any vertex v and pointer Q, $Q(v)$ can be "followed" in $O(1)$ time using the labels of v and $P(v)$.
  - A pointer to the blocks immediately to the left and right of B, denoted by $I_L(B)$ and $I_R(B)$, respectively. Both $I_L(B)$ and $I_R(B)$ are artificial constructs, as they are achieved by including I_L and I_R pointers in the label of $P(B)$, as per the pointer technique described above. These pointers add $O(\mathcal{L})$ bits to the label of the pointer vertex only.
  - A pointer to the furthest adjacent blocks to the left and right of B, denoted by $F_L(B)$ and $F_R(B)$, respectively. These pointers are achieved using the pointer technique described above, and add $O(\mathcal{L})$ bits to the label of the pointer vertex only.
  - The size of B, denoted $s(B)$. This value is kept in the label of the pointer vertex, adding $O(\log n)$ bits to its label.

We observe that the total size for each label is $O(\mathcal{L} + \mathcal{B}) \in O(\log n)$. Furthermore, we can determine the adjacency of two vertices u and v in $O(1)$ time, using only their labels, by checking if $f_L(v) \leq b(u) \leq f_R(v)$.

In comparison to the data structure used by Hell et al., our vertex labels do not include self pointers or end pointers. Self pointers become obsolete in our vertex centered setting, and end pointers, although useful, have proven difficult to maintain. Furthermore, for every block B, Hell and al. point to the adjacent blocks immediately to the left and right of B, whereas, we include a similar pointer that omits the adjacency condition. By dropping the adjacency condition we are able to maintain additional information about the straight enumeration without sacrificing asymptotic space or running times.

Although it is much easier to discuss pointers and values at a block level, we must always ensure that these items can be observed at the vertex level. For instance, a vertex v is an end vertex if and only if $F_L(B(v)) = B(v)$ or $F_R(B(v)) = B(v)$. However, to determine this condition, we must check to see if $f_L(v) = b(v)$ or $f_R(v) = b(v)$. As such, in the ensuing discussion we maintain the convention of offsetting the vertex level condition in square brackets, for example, "$F_L(B(v)) = B(v)$ $[f_L(v) = b(v)]$". Nearly all of the conditions mentioned herein can be tested in O(1) time. As such, we will only comment on the time required to check a condition if it takes $\omega(1)$ time to check the condition.

When discussing such conditions, we maintain the convention that pointers acting on a block produce a block, and pointers acting on a vertex produce a vertex. For example, $F_L(B(v))$ is a block, whereas, $F_L(v)$ is a vertex in the block $F_L(B(v))$. As previously discussed, $F_L(B(v))$ is an artificial construct.

Although we have given significant consideration to the labels of the dynamic scheme, we have not yet discussed the marker. Deng, Hell, and Huang [14] provide an $O(n+m)$ time algorithm for generating a straight enumeration of a proper interval graph from, presumably, an adjacency list representation (actually, their algorithm presents a vertex ordering, but minor bookkeeping will give a straight enumeration of blocks). Where $\mathbf{B}$ is the number of blocks in the straight enumeration, we can use the straight enumeration to establish the $\mathbf{B}$ circular doubly linked lists in $\Theta(n)$ time. Next, establishing pointer vertices and block identifiers, as well as the b, f_L, and f_R values requires a further $O(n)$ time. Finally, establishing the various pointers requires an additional $\Theta(\mathbf{B}) \in O(n)$ time. Therefore, if provided with the straight enumeration, the marker requires $\Theta(n)$ time; otherwise, the marker requires $O(n+m)$ time.

6.1.1 Relabeller

For the most part, we will discuss the relabelling algorithm at the block level, including the vertex-level discussion in Appendix C. In the ensuing discussion, G is the original graph and G' is the new graph.

Given the linear nature of the straight enumeration, the limiting factor inherent in our labelling scheme is the maintenance of the b, f_L and f_R values. When the graph is modified, our first task is to modify blocks, pointer vertices, and pointers, as necessary, in order to maintain a straight enumeration. Once this is complete, we can traverse I_L and I_R pointers to determine the entire straight enumeration. Knowing the entire straight enumeration, one pass through the ordering (from least to greatest) is sufficient to re-assign optimal block identifiers by traversing the circular linked list of vertices in each block. Having assigned these optimal block identifiers, a second pass is sufficient to assign the f_L and f_R values, which depend on the block identifiers, to the vertices in each block.

Regardless of the graph operation under consideration, the maintenance of the b, f_L, and f_R values takes as much as $\Theta(n)$ time. Because this approach can be used to maintain optimal b values, we did not employ the assumption on the size of the identifiers, as stated in Section 3.1.3, to $\mathcal{B}$, the size of the largest block identifier.

Unlike our work with r-minoes and r-bics, seen in Chapter 5, we will not use the existence of an error-detection dynamic adjacency labelling scheme to establish a recognition result, even though proper interval graphs are hereditary. As discussed, our relabeller can take as much as $\Theta(n)$ time to handle vertex addition, therefore, the recognition time offered by Theorem 3.7 could be as high as $\Theta(n^2)$. In comparison, Corneil et al. [12], among others, have already established $O(n + m)$ recognition of proper interval graphs.

Deleting a vertex

Let v be the vertex to be deleted, where X denotes the neighbourhood of v in G. As well, let the contig containing $B(v)$ be $B_1 \preceq \ldots \preceq B_i \preceq \ldots \preceq B_l \preceq \ldots \preceq B_j \preceq \ldots \preceq B_k$, where $B_l = B(v)$, $B_i = F_L(B_l)$, and $B_j = F_R(B_l)$. The action of the relabeller depends on whether $B_l = \{v\}$ $[nx(v) = v]$.

The following lemma addresses how the straight enumeration changes when $B_l \neq \{v\}$.

Lemma 6.8 *Let G' be a proper interval graph formed by deleting a vertex v from a proper interval graph G, where $B(v) \neq \{v\}$. If B' is a block in G', then either B' is a block in G, or $B' \cup \{v\}$ is a block in G.*

Proof. Assume the contrary, that is, neither B' nor $B' \cup \{v\}$ is a block in G. Since B' is not a block in G, one of the following statements must be hold.

- There exists b_1 and b_2 in B' for which $N_G[b_1] \neq N_G[b_2]$.
- There exists some b^* in $V_G \setminus B'$ for which $N_G[b^*] = N_G[b]$, for all b in B', where N_G denotes the neighbourhood in G (as opposed to G').

If the former condition holds, then consider that the only vertex adjacencies which change from G to G' are those with v. Since $N_{G'}[b_1] = N_{G'}[b_2]$, exactly one of $N_G[b_1]$ and $N_G[b_2]$ contains v. Without loss of generality, assume that $v \in N_G[b_1]$. Since $B(v) \neq \{v\}$, there exists some additional vertex v^* in $B(v)$. Therefore, $v^* \in N_G[b_1]$, but $v^* \notin N_G[b_2]$. Since $v^* \neq v$, $v^* \in N_{G'}[b_1]$, yet $v^* \notin N_{G'}[b_2]$, so B' is not a block in G', which is a contradiction.

Assuming the latter condition holds, we need to consider two cases. First, if $b^* = v$, then consider that $B' \subseteq B(v) \setminus \{v\}$ as $N_G[v] = N_G[b]$, for all b in B'. Moreover, for all v^* in $B(v) \setminus \{v\}$ and for all b in B', $N_{G'}[v^*] = N_G[v^*] \setminus \{v\} = N_G[b] \setminus \{v\} = N_{G'}[b]$. Therefore, $B(v) \setminus \{v\} \subseteq B'$, which gives $B' = B(v) \setminus \{v\}$. That is, $B' \cup \{v\}$ is a block of G, which is a contradiction.

Secondly, if $b^* \neq v$, then $b^* \in V_{G'}$. But $b^* \notin B'$, where B' is a block of G', therefore, $N_{G'}[b^*] \neq N_{G'}[b]$, for all b in B'. Specifically, there exists some $b^{**} \in N_{G'}[b^*]$ for which $b^{**} \notin N_{G'}[b]$, for all b in B', where $b^{**} \neq v$. Therefore, $N_G[b^*] \neq N_G[b]$, for all b in B', which is a contradiction. □

Where B_l contains another vertex besides v, Lemma 6.8 tells us that a straight enumeration for G' can be obtained from a straight enumeration of G by removing v from B_l. This scenario is depicted in Figure 6.3. Specifically, our labelling is amended as follows.

- Remove all references to v.
  - We must change all references to v as a pointer vertex. Specifically, if $v = P(v)$, then we make $nx(v)$ the pointer vertex by changing its label to reflect the pointers, and changing the labels of all the vertices in B_l to reflect that $nx(v)$ is the new pointer vertex. This change can be done in $O(|B_l|) \in O(|X|)$ time by traversing B_l, beginning at v. Let q be the resulting pointer vertex of B_l.
  - We must change all references to v in I_L and I_R pointers. Providing $I_L(B_l) \neq \text{NIL}$ $[I_L(q) \neq \text{NIL}]$, set $I_R(I_L(q))$ to q. Similarly, providing $I_R(B_l) \neq \text{NIL}$, set $I_L(I_R(q))$ to q. These changes take $O(1)$ time.
  - We must change all references to v in F_L and F_R pointers. Specifically, for any block B, if $F_L(P(B))$ or $F_R(P(B))$ is v, then we change its value to q. Now, if $F_R(P(B)) = v$, then, by Lemma 6.7 (umbrella property), $B_i \preceq B \preceq B_l$; similarly, if $F_L(P(B)) = v$, then $B_l \preceq B \preceq B_j$. As such, we can recursively follow I_L and I_R pointers to determine all such blocks B. These changes take $O(deg(B_l)) \in O(|X|)$ time.
  - We must remove v from the circular doubly linked list of the vertices in B_l. This removal takes $O(1)$ time.
- Decrease the value of $s(B_l)$ $[s(q)]$ by one. This operation takes $O(1)$ time.
- Delete v. This deletion takes $O(1)$ time.

The following lemma addresses how the straight enumeration changes when $B_l = \{v\}$.

Lemma 6.9 *Let G' be a proper interval graph formed by deleting a vertex v from a proper interval graph G, where $B(v) = \{v\}$. If B' is a block in G', then either B' is a block in G, or $B' = B_\alpha \cup B_\beta$, where B_α and B_β are blocks in G.*

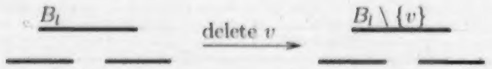


Figure 6.3: Deleting a vertex v, where $B_l = B(v) \neq \{v\}$

Proof. Assume the contrary, that is, B' is not a block of G and $B' \neq B_\alpha \cup B_\beta$, where B_α and B_β are distinct blocks in G. Since B' is not a block in G, one of the following statements must be hold.

- There exists b_1 and b_2 in B' for which $N_G[b_1] \neq N_G[b_2]$.
- There exists some b^* in $V_G \setminus B'$ for which $N_G[b^*] = N_G[b]$, for all b in B'.

If the latter condition holds, arguments identical to those seen in the proof of Lemma 6.8 will lead to contradictions. However, there is a difference in the contradiction achieved when $b^* = v$, as $B' \subseteq B(v) \setminus \{v\}$ gives $B' = \emptyset$.

Assuming the former condition holds, then consider that the only vertex adjacencies which change from G to G' are those with v. Since $N_{G'}[b_1] = N_{G'}[b_2]$, exactly one of $N_G[b_1]$ and $N_G[b_2]$ contains v. Without loss of generality, assume that $v \in N_G[b_1]$. Moreover, since $N_{G'}[b]$ is unique for all b in B', the uniqueness of v allows us to partition the members of B' into B'_1 and B'_2, where $b \in B'_1$ if and only if $v \in N_G[b]$.

Since $N_{G'}[b]$ is unique for all $b \in B'_2$, so too is $N_G[b]$. Therefore, $B'_2 \subseteq B_\alpha$, for some block B_α of G. Observe that $B_\alpha \neq B(v)$, otherwise, $B(v) = \{v\}$ contradicts the existence of b_2. Now, for all b_α in B_α, $N_G[b_\alpha] = N_{G'}[b_\alpha] = N_{G'}[b_2] = N_G[b_2]$, so $B_\alpha \subseteq B'_2$. Therefore, $B'_2 = B_\alpha$.

Since $N_{G'}[b]$ is unique for all $b \in B'_1$, so too is $N_G[b]$. Therefore, $B'_1 \subseteq B_\beta$, for some block B_β of G. Observe that $B_\beta \neq B(v)$, otherwise, $B(v) = \{v\}$ contradicts the existence of b_1. Furthermore, observe that $B_\beta \neq B_\alpha$, as $N_G[b_1] \neq N_G[b_2]$. Now, for all b_β in B_β, $N_G[b_\beta] = N_{G'}[b_\beta] \cup \{v\} = N_{G'}[b_1] \cup \{v\} = N_G[b_1]$, so $B_\beta \subseteq B'_1$. Therefore, $B'_1 = B_\beta$ and the result follows. □

From Lemma 6.9, we see that a block of G' can be formed by merging two blocks of G. The following theorem addresses which two blocks are merged.

Lemma 6.10 *Let G' be a proper interval graph formed by deleting a vertex v from a proper interval graph G, where $B(v) = \{v\}$. If $B' = B_\alpha \cup B_\beta$ is a block in G', where B_α and B_β are distinct blocks in G, then $\{B_\alpha, B_\beta\}$ is either $\{B_{i-1}, B_i\}$ or $\{B_j, B_{j+1}\}$.*

Proof. First, let us assume that $B_\beta \prec B_l$. Since the neighbours of B_α and B_β will be identical upon deletion of v, $F_L(B_\alpha) = F_L(B_\beta)$, and either $F_R(B_\beta) = B_l$ and $F_R(B_\alpha) = B_{l-1}$, or vice versa; without loss of generality, we assume the former. By Lemma 6.7

(umbrella property), $B_\alpha \prec B_i$ and $B_i \preceq B_\beta$. Now if $B_i \prec B_\beta$, then $F_R(B_\beta) = B_l$ combined with Lemma 6.7 (umbrella property), gives $F_R(B_i) = B_l = F_R(B_\beta)$. Therefore, by Lemmas 6.4 and 6.7 (umbrella property), $F_L(B_i) \prec F_L(B_\beta)$. Yet, by Lemma 6.7 (umbrella property), $F_L(B_\alpha) \preceq F_L(B_i)$, thereby, $F_L(B_\alpha) \neq F_L(B_\beta)$, which is a contradiction. Therefore, $B_\beta = B_i$.

Furthermore, $F_R(B_\alpha) = B_{l-1}$ and $F_L(B_l) = B_i$ give $F_R(B_{i-1}) = B_{l-1} = F_R(B_\alpha)$. Yet, $B_\alpha \preceq B_{i-1} \prec B_\beta$ and $F_L(B_\alpha) = F_L(B_\beta)$, give $F_L(B_\alpha) = F_L(B_{i-1})$. Thereby, $N[B_\alpha] = N[B_{i-1}]$, which is to say that $B_\alpha = B_{i-1}$.

Using similar arguments we can show that, for $B_l \prec B_\beta$, $B_\beta = B_j$ and $B_\alpha = B_{j+1}$. □

Where B_l contains only v, Lemma 6.9 can be used to obtain a straight enumeration of G'. We remove all references to v, and merge blocks as per Lemma 6.10, as depicted in Figure 6.4. Recall that two blocks are merged if and only if the deletion of v causes their neighbourhoods to be the same.

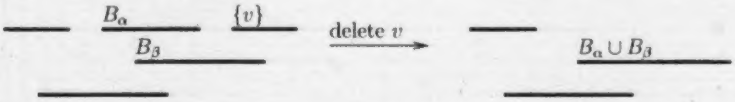


Figure 6.4: Merging blocks, where the vertex v is deleted from the graph

Specifically, our labelling is changed as follows.

- If $1 < i < l$ $[f_L(v) \neq f_L(F_L(v))$ and $f_L(v) \neq b(v)]$, $F_L(B_{i-1}) = F_L(B_i)$ $[f_L(I_L(F_L(v))) = f_L(F_L(v))]$, and $F_R(B_{i-1}) = B_{l-1}$ $[f_R(I_L(F_L(v))) = b(I_L(v))]$, then merge B_i into B_{i-1}.
  - Add the value of $s(B_i)$ to $s(B_{i-1})$ [add $s(P(F_L(v)))$ to $s(P(I_L(F_L(v))))$]. This operation takes O(1) time.
  - Set $I_R(B_{i-1})$ to B_{i+1} $[I_R(I_L(F_L(v)))$ to $I_R(F_L(v))]$ and $I_L(B_{i+1})$ to B_{i-1} $[I_L(I_R(F_L(v)))$ to $I_L(F_L(v))]$. These assignments take O(1) time.
  - Update the labels of the vertices of B_i to reflect the fact that $P(B_{i-1})$ $[P(I_L(F_L(v)))]$ is the pointer vertex of the merged block. This update can be done in $O(|B_i|) \in O(|X|)$ time by traversing B_i, beginning at $F_L(v)$.
  - Merge the two circular doubly linked lists, using $P(B_{i-1})$ $[P(I_L(F_L(v)))]$ and $P(B_i)$ $[P(F_L(v))]$ as reference points. This merge takes O(1) time.
- If $l < j < k$, $F_R(B_j) = F_R(B_{j+1})$, and $F_L(B_{j+1}) = B_{l+1}$ then merge B_j into B_{j+1}. This merge takes $O(|B_j|) \in O(|X|)$ time.

- Providing $I_L(B_l) \neq \text{NIL}$ $[I_L(v) \neq \text{NIL}]$, set $I_R(I_L(B_l))$ to $I_R(B_l)$ $[I_R(I_L(v))$ to $I_R(v)]$. Similarly, providing $I_R(B_l) \neq \text{NIL}$, set $I_L(I_R(B_l))$ to $I_L(B_l)$. These assignments take O(1) time.

- For each block B in $\{B_i, \ldots, B_{l-1}\}$, if $F_R(B) = B_l$ $[f_R(P(B)) = b(v)]$, then set $F_R(B)$ to B_{l-1} $[F_R(P(B))$ to $I_L(v)]$. As well, for each block B in $\{B_{l+1}, \ldots, B_j\}$, if $F_L(B) = B_l$, then set $F_L(B)$ to B_{l+1}. These assignments can be done in $O(deg(B_l)) \in O(|X|)$ time, by recursively following I_L and I_R pointers to determine all such blocks B.

- Delete v. This deletion takes O(1) time.

- Relabel the blocks and adjust the b, f_L, and f_R values as previously discussed. Given that there can be as many as $\Theta(n)$ blocks, this operation can take as much as $O(n)$ time.

The correctness, at least the block level, of our algorithm is due to the correctness of Lemmas 6.8, 6.9, and 6.10, and the exhaustiveness of the cases considered. Ensuring that the algorithm does what we want it to do, in terms of pointers and vertex labels, is a matter of verification.

For the remaining graph operations: adding a vertex, deleting an edge, and adding an edge, we will not present the same level of rigour as seen in Lemmas 6.8, 6.9, and 6.10, unless the change to the straight enumeration is not obvious. However, for each of these remaining operations, we will be careful to enumerate all possible cases, including those which prevent G' from being a proper interval graph.

Adding a vertex

Let v be the vertex to be added, where X denotes the neighbourhood of v in G'. We will say that v is adjacent to a block B if $B \cap X \neq \emptyset$, fully adjacent to B if $B \subseteq X$, and partially adjacent to B if it is adjacent, but not fully adjacent.

Given a straight enumeration Φ of a connected proper interval graph G, Φ can be thought of as a weak linear order $\prec_\Phi$ of V_G, where $v_1 \prec_\Phi v_2$ if and only if $B(v_1) \prec B(v_2)$ in Φ. Hell et al. [27] say that Φ' is a *refinement* of Φ, if for every $v_1, v_2 \in V_G$, $v_1 \prec_\Phi v_2 \Longrightarrow v_1 \prec_{\Phi'} v_2$, or for every $v_1, v_2 \in V_G$, $v_1 \prec_\Phi v_2 \Longrightarrow v_2 \prec_{\Phi'} v_1$. Observe that in the latter case, ${\Phi'}^R$ is also a refinement, where for every $v_1, v_2 \in V_G$, $v_1 \prec_\Phi v_2 \Longrightarrow v_1 \prec_{\Phi'} v_2$.

The following lemma partially addresses how the straight enumeration of G' compares to the straight enumeration of G, when G' is a proper interval graph.

Lemma 6.11 *[27] If G is a connected induced subgraph of a proper interval graph H, where Φ_G is a contig of G and Φ_H is a straight enumeration of H, then Φ_H is a refinement of Φ_G.*

In essence, Lemma 6.11 tells us that there is a straight enumeration of G' that is much like the straight enumeration of G, except that some blocks are further "refined". To specifically address this "refinement" of certain blocks, Hell et al. observe that whenever v is partially adjacent to a block B of G, B will be split into $B \cap N(v)$ and $B \setminus N(v)$ in G'. Furthermore, they argue that, in any straight enumeration of G', $B \cap N(v)$ and $B \setminus N(v)$ occur consecutively. As well, they observe that whenever v is not partially adjacent to a block B of G, either B or $B \cup \{v\}$ will be a block of G'.

To summarize, if G' is a proper interval graph, then it has some straight enumeration which looks much like that of G but with some blocks split and, possibly, one block which now has v added to it. In determining exactly when the addition of v maintains a proper interval graph, we consider the following lemmas.

Lemma 6.12 *[27] Let G be a proper interval graph. If $G + v$ is a proper interval graph then v can have neighbours in at most two components of G.*

Lemma 6.13 *Consider a proper interval graph G, to which a new vertex v is added. Let C be a component of G containing vertices of $N(v)$ and let $\{B_1, \ldots, B_k\}$ be the set of blocks in C containing members of $N(v)$, where, in a contig of C, $B_1 \prec \ldots \prec B_k$. If $G + v$ is a proper interval graph, then the following properties are satisfied.*

1. *[27] $B_1, \ldots, B_k$ are consecutive in the contig of C.*
2. *[27] If $k \geq 3$, then v is fully adjacent to $B_2, \ldots, B_{k-1}$.*
3. *If $k \geq 3$, then $\{B_1, \ldots, B_k\}$ does not contain three pairwise non-adjacent blocks.*
4. *If $k = 2$, then v must be fully adjacent to at least one of B_1 and B_2.*
5. *[27] If v is adjacent to a single block B_1 in C, then B_1 is an end block.*
6. *[27] If v is adjacent to more than one block in C and has neighbours in another component, then B_1 is adjacent to B_k, and one of B_1 or B_k is an end block to which v is fully adjacent, while the other is an inner block.*

Proof. (of condition 3) Assume the contrary, that is, $\{B_1, \ldots, B_k\}$ contains three pairwise non-adjacent blocks B_{i_1}, B_{i_2}, and B_{i_3}. For each B_j, $1 \leq j \leq k$, let v_j be a vertex in block $B_j \cap N(v)$. In $G + v$, the induced graph on $\{v, v_{i_1}, v_{i_2}, v_{i_3}\}$ forms an induced $K_{1,3}$, therefore, $G + v$ is not a proper interval graph.

(of condition 4) Assume that v is fully adjacent to neither B_1 nor B_2. As such, consider vertices b_1 and b_2, from B_1 and B_2, respectively, to which v is adjacent, and vertices b_1' and b_2', from B_1 and B_2, respectively, to which v is not adjacent.

From condition 1 of Lemma 6.13, we know that B_1 and B_2 are adjacent. From Lemma 6.5, we know that B_1 cannot have the same neighbours as B_2. Without loss of generality, assume that B_1 is a adjacent to some block B to which B_2 is not adjacent. By Lemma 6.7 (umbrella property), we know that $B \prec B_1$, so v is not adjacent to B. Where b is a member of B, the induced graph on $\{b_1, v, b_2', b\}$ is a $K_{1,3}$. Therefore, $G + v$ is a not a proper interval graph. □

For any vertex addition, we claim that we can determine in $O(n)$ time whether Lemmas 6.12 and 6.13 are satisfied. Before describing how this is done, consider the following operations of which we can avail.

- Given a set of vertices S and a vertex u, we can determine in $O(|S|)$ time whether $B(u) \subseteq S$ by traversing $B(u)$, beginning at u, until we find a vertex not in S. Moreover, given vertices $u_1, \ldots, u_k$ belonging to distinct blocks $B_1, \ldots, B_k$, we can use this approach to determine which blocks are subsets of S in $O(k + |S|)$ time. If each u_i belongs to S, then this time reduces to $O(|S|)$.
- Given a set of vertices S and vertices $u_1, \ldots, u_k$ belonging to distinct blocks $B_1, \ldots, B_k$, for which $B_i \subseteq S$, we can determine $S \setminus (\cup B_i)$ in $O(\Sigma|B_i|) \in O(|S|)$ time by traversing each B_i, beginning at u_i.
- Given a set of vertices S and a vertex u, we can determine $S \setminus B(u)$ and $S \cap B(u)$ in $O(|S|)$ time by comparing the b value of each vertex in S with that of u.

The aforementioned operations are used by the algorithm LEFTCOMPONENTBLOCKSTRUCTURE, shown in Figure 6.5, to evaluate the conditions of Lemmas 6.12 and 6.13. LEFTCOMPONENTBLOCKSTRUCTURE first examines X to determine a vertex v_1 whose block, B_1, is the leftmost of all blocks containing members of X; that is, v_1 has the minimal b value over all vertices in X. The vertex v_1, and hence B_1, can be determined in $O(|X|)$ time. Regarding B_1, we make note of the following.

- The vertex v_1.
- The adjacency of v, full or partial, with B_1. As discussed previously, we can determine whether $B_1 \subseteq X$ in $O(|X|)$ time.
- Whether B_1 is an end block $[f_L(v_1) = b(v_1)$ or $f_R(v_1) = b(v_1)]$.
- The vertices in $X \cap B_1$. As discussed previously, this set can be determined in $O(|X|)$ time.

Let C_1 denote the component containing v_1. Provided $X \setminus B_1 \neq \emptyset$ and $F_R(B_1) \neq B_1$ $[f_R(v_1) \neq b(v_1)]$, we obtain similar information about $B_2 = I_R(B_1)$ (using $v_2 = I_R(v_1)$ as

LeftComponentBlockStructure(G, X)

Input: An adjacency labelling of a graph G created using our dynamic scheme, and a subset X of V_G.

Output: Let Φ be the straight enumeration of G employed in the dynamic scheme, and let C be the leftmost component in Φ containing a vertex in X. Furthermore, where $G + v$ is the graph formed by adding a new vertex v to G such that v is adjacent to exactly those vertices in X, let the blocks of C be denoted as per the hypothesis of Lemma 6.13.
Providing G satisfies conditions 1 through 3 of Lemma 6.13, the algorithm outputs certain information about the structure of G. If $G + v$ does not satisfy these criteria, the output indicates as such.

```
$v_1 \leftarrow$ a leftmost vertex in $X$
$B_1 \leftarrow$ the block containing $v_1$
$clawblock \leftarrow B_1$
$clawcount \leftarrow 1$
$i \leftarrow 1$
$endblock \leftarrow 0$
$adjacent \leftarrow 1$
while $X \setminus \cup_{j=1}^{i-1} B_j \neq \emptyset$ and $endblock = 0$ and $adjacent = 1$ do
    if $v$ is not adjacent to $B_i$ then
        if $C \cap (X \setminus \cup_{j=1}^{i-1} B_j) \neq \emptyset$ then
            error no longer a proper interval graph
        else $i = i - 1$
            $adjacent \leftarrow 0$
    elseif $i \geq 3$ and $v$ is not fully adjacent to $B_{i-1}$ then
        error no longer a proper interval graph
    else record the vertex $v_i$
        record the adjacency (full or partial) of $v$ with $B_i$
        record whether $B_i$ is an end block
        record $(X \setminus \cup_{j=1}^{i-1} B_j) \cap B_i$
        if $B_i$ is not adjacent to $clawblock$ then
            $clawblock \leftarrow B_i$
            $clawcount \leftarrow clawcount + 1$
            if $clawcount = 3$ then
                error no longer a proper interval graph
        if $F_R(B_i) = B_i$ then
            $endblock \leftarrow 1$
        else $i \leftarrow i + 1$
            $v_i \leftarrow I_R(v_{i-1})$
            $B_i \leftarrow$ the block containing $v_i$
record the value of $k$ as $i$
record the set $Y$
```

Figure 6.5: The algorithm LeftComponentBlockStructure used to test the criteria of Lemmas 6.12 and 6.13

a starting point for the vertices in B_2); however, we now need to know if $X \setminus \cup_{j=1}^{2} B_j \neq \emptyset$. This calculation is done in the order $(X \setminus B_1) \setminus B_2$, where $X \setminus B_1$ has been determined in $O(|X|)$ time, as discussed above. This process is repeated, setting $v_i = I_R(v_{i-1})$ and $B_i = B(v_i) = I_R(B_{i-1})$ until one of the following occurs.

- **Termination event 1** (v is not adjacent to B_i, $X \setminus \cup_{j=1}^{i} B_j \neq \emptyset$, and there are vertices in $X \setminus \cup_{j=1}^{i} B_j$ that belong to C_1): In this case, condition 1 of Lemma 6.13 is violated.

  Exactly how to determine if there are vertices in $X \setminus \cup_{j=1}^{i} B_j$ that belong to C_1 requires some consideration. Specifically, we traverse the F_R pointers from B_i to determine the other end block B_{end} of C_1, then check to see if there is some vertex u in $X \setminus (\bigcup_{j=1}^{i} B_j)$, for which $B(u) \prec B_{end}$ [$b(u) < b(v_{end})$, where v_{end} is a vertex of B_{end}]. Checking each vertex u in $X \setminus (\bigcup_{j=1}^{i} B_j)$ to see if $B(u) \prec B_{end}$ takes only $O(|X \setminus (\bigcup_{j=1}^{i} B_j)|) \in O(|X|)$ time, however, it takes as much as $\Theta(n)$ time to traverse the F_R pointers to determine B_{end}.

- **Termination event 2** ($i \geq 3$, v is adjacent to B_i, and v is not fully adjacent to B_{i-1}): In this case, condition 2 of Lemma 6.13 is violated.

- **Termination event 3** (the variable *clawcount* has value 3): In this case, v is adjacent to three blocks which are pairwise non-adjacent. Therefore, condition 3 of Lemma 6.13 is violated.

- **Termination event 4** (v is not adjacent to B_i, $X \setminus \cup_{j=1}^{i} B_j \neq \emptyset$, and there are no vertices in $X \setminus \cup_{j=1}^{i} B_j$ that belong to C_1): In this case, the component C_1 containing v_1 satisfies conditions 1 through 3 of Lemma 6.13, otherwise, the selection of B_i as $I_R(B_{i-1})$ guaranteed that we would have encountered one of the first three termination events. Moreover, we can test conditions 4 and 5 in $O(1)$ time, as we have noted the value of k, the adjacency of v with B_1 and B_k, and whether B_1 is an end block.

  Of greater interest, however, is the fact that not all the vertices of X belong to the same component. As such, we must also confirm that C_1 satisfies condition 6 of Lemma 6.13. Given that we have noted the vertices v_1 and v_k, the adjacency of v with B_1 and B_k, and whether B_1 and B_k are end blocks, we can test this condition in $O(1)$ time.

  As well, we must verify that the vertices $X \setminus \cup_{j=1}^{k} B_j$ belong to exactly one component, C_2, that satisfies conditions 1 through 6 of Lemma 6.13. We verify these criteria by using LEFTCOMPONENTBLOCKSTRUCTURE on the set $X \setminus \cup_{j=1}^{k} B_j$.

- **Termination event 5** ($X \setminus \cup_{j=1}^{i} B_j \neq \emptyset$, but $F_R(B_i) = B_i$): In this case, the vertices of X belong to more than one component, however, $F_R(B_i) = B_i$ gives that there are no vertices in $X \setminus \cup_{j=1}^{i} B_j$ that belong to the component containing v_1. As such, this termination event is handled in the same manner as termination event 4.

- **Termination event 6** ($X \setminus \cup_{j=1}^{i} B_j = \emptyset$): In this case, the vertices of X all belong to the same component. The component C_1 containing v_1 satisfies conditions 1 through 3 of Lemma 6.13, otherwise, we would have encountered one of the first three termination events. Moreover, we can test conditions 4 and 5 in O(1) time, as we have noted the value of k, the adjacency of v with B_1 and B_k, and whether B_1 is an end block. Lemma 6.12 and condition 6 of Lemma 6.13 are not applicable.

Given that C_1 is finite, termination events 5 and 6 guarantee that the algorithm LEFTCOMPONENTBLOCKSTRUCTURE will terminate. The exhaustiveness of the termination events, along with the careful consideration of their connections to Lemmas 6.12 and 6.13, ensure that we can determine whether Lemmas 6.12 and 6.13 are satisfied.

Now consider that termination event 2 guarantees that there are at most three values of i for which B_i is examined and found not fully adjacent to v, namely, B_1, B_k and B_{k+1}. As discussed, each of these adjacencies can be determined in $O(|X|)$ time. Furthermore, $X \setminus B_1$ can be determined in $O(|X|)$ time. For $k \geq 3$, each of $B_2, \ldots, B_{k-1}$ is a subset of $X \setminus B_1$ (each of these blocks is fully adjacent to v); therefore, $(X \setminus B_1) \setminus \cup_{j=2}^{k-1} B_j = X \setminus \cup_{j=1}^{k-1} B_j$ can also be determined in $O(\Sigma_{j=2}^{k-1}|B_j|) \in O(|X \setminus B_1|) \in O(|X|)$ time. Finally, $(X \setminus \cup_{j=1}^{k-1} B_j) \setminus B_k = X \setminus \cup_{j=1}^{k} B_j$ can be determined in $O(|X \setminus \cup_{j=1}^{k-1} B_j|) \in O(|X|)$ time. That is, the total time required to determine $X \setminus \cup_{j=1}^{k} B_j$, from X is $O(|X|)$.

Unfortunately, the running time of LEFTCOMPONENTBLOCKSTRUCTURE is dominated by the possible $\Theta(n)$ time required to distinguish between termination events 1 and 3. Specifically, the running time of LEFTCOMPONENTBLOCKSTRUCTURE, hence, the time required to verify the conditions of Lemmas 6.12 and 6.13 could be as much as $\Theta(n)$.

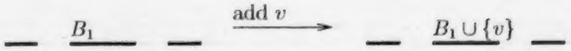
Hereafter, we assume that Lemmas 6.12 and 6.13 are satisfied by our vertex addition. Note that, while verifying that Lemmas 6.12 and 6.13 are satisfied, we have recorded a great deal about the structure of the blocks. This information will be used to help us relabel the vertices.

In describing the relabelling, let us first consider when the members of X belong to one component, C. As in the hypothesis of Lemma 6.13, let $\{B_1, \ldots, B_k\}$ denote the set of blocks in C that are adjacent to v, such that in the contig of C, $B_1 \prec \ldots \prec B_k$. We consider three cases, depending on the value of k.

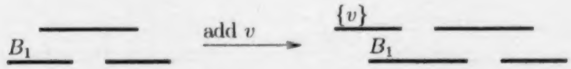
1. $k = 1$. By Lemma 6.13, B_1 is an end block. Without loss of generality, assume that $B_1 \prec B$, for any block B in C.

   If v is fully adjacent to B_1, and $C = B_1$ [$f_R(v_1) = b(v_1)$], then we add v to B_1, as depicted in Figure 6.6(a). If v is fully adjacent to B_1, but $C \neq B_1$, then we add the block $\{v\}$ immediately before B_1, as depicted in Figure 6.6(b). Finally, if v is not fully adjacent to B_1, then we partition $B_1 \cup \{v\}$ into $\{v\} \prec X \prec B_1 \setminus X$, as depicted in

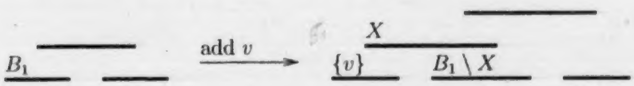
Figure 6.6(c).



(a) v is fully adjacent to B_1, and the vertices of B_1 form a component



(b) v is fully adjacent to B_1, and the vertices of B_1 do not form a component



(c) v is not fully adjacent to B_1

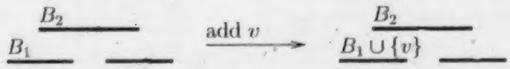
Figure 6.6: Adding the vertex v, where v is adjacent only to B_1. Without loss of generality, B_1 is assumed to be the leftmost block in its component

2. $\underline{k = 2.}$ By condition 4 of Lemma 6.13, v must be fully adjacent to at least one of B_1 and B_2. Without loss of generality, assume that v is fully adjacent to B_1. Let $B_i = F_L(B_1)$ and $B_j = F_R(B_1)$.

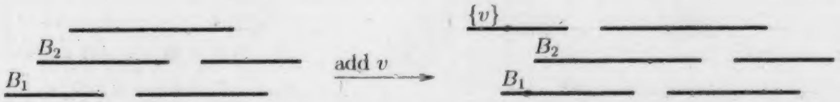
   Assume first that $B_i = B_1$ [$f_L(v_1) = b(v_1)$]. Note that Lemma 6.4 guarantees that $B_2 \prec F_R(B_2)$. If v is fully adjacent to B_2 and $B_j = B_2$ [$f_R(v_1) = b(v_2)$], then add v to B_1, as shown in Figure 6.7(a). If v is fully adjacent to B_2, but $B_2 \prec B_j$ [$b(v_2) = f_R(v_1)$], then add the block $\{v\}$ immediately before B_1, as shown in Figure 6.7(b). Finally, if v is not fully adjacent to B_2, then we partition $B_1 \cup B_2 \cup \{v\}$ into $\{v\} \prec B_1 \prec B_2 \cap X \prec B_2 \setminus X$, as shown in Figure 6.7(c).

   Now assume that $B_i \prec B_1$ [$f_L(v_1) < b(v_1)$] As such, if G' is to be a proper interval graph, then the block containing v in the straight enumeration of G' must be ordered to the right of any resultant block of G' that contains a member of B_1; for simplicity, we indicate this by saying that, $B_1 \prec B(v)$. If $B_2 \prec B_j$ [$b(v_2) < f_R(v_1)$], then $B(v) \prec B_1$; this contradiction tells us that G' is not a proper interval graph. Consequently, assume that $B_j \preceq B_2$, which is to say that $B_j = B_2$ [$f_R(v_1) = b(v_2)$]. Furthermore, if v is not fully adjacent to B_2, then B_1 adjacent to B_2 gives $B_2 \prec B(v)$, yet, with v not fully adjacent to B_2 and v adjacent to B_1, $B(v) \prec B_2$, which is another contradiction. Therefore, assume that v is also fully adjacent to B_2.

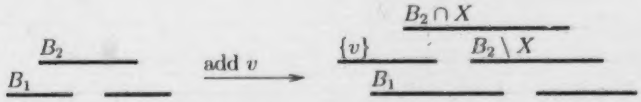
   Given $B_i \prec B_1$, $B_j = B_2$, and v fully adjacent to B_2, if $F_L(B_2) \prec B_1$ [$f_L(v_2) = b(v_1)$],



(a) v is fully adjacent to B_2, and $F_R(B_1) = B_2$



(b) v is fully adjacent to B_2, and $B_2 \prec F_R(B_1)$



(c) v is not fully adjacent to B_2

Figure 6.7: Adding the vertex v, where v is adjacent only to B_1 and B_2 and $F_L(B_1) = B_1$. Without loss of generality, we have assumed that v is fully adjacent to B_1

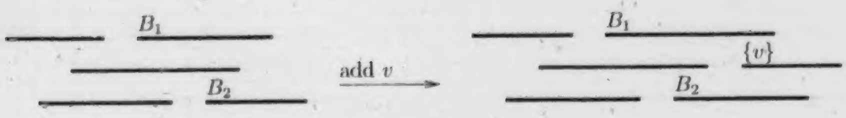
then $B_2 \prec B(v)$. If, furthermore, $B_2 \prec F_R(B_2)$ $[b(v_2) < f_R(v_2)]$, then $B(v) \prec B_2$, which is a contradiction. On the other hand, if $F_R(B_2) = B_2$ $[f_R(v_2) = b(v_2)]$, then we add the block $\{v\}$ immediately after B_2, as depicted in Figure 6.8(a).

Finally, given $B_i \prec B_1$, $B_j = B_2$, v fully adjacent to B_2, and $F_L(B_2) = B_1$, if $F_R(B_2) = B_2$ $[f_R(v_2) = b(v_2)]$, then we need only add v to B_2, as depicted in Figure 6.8(b). However, if $B_2 \prec F_R(B_2)$ $[b(v_2) < f_R(v_2)]$, then we add the block $\{v\}$ between B_1 and B_2, as depicted in Figure 6.8(c).

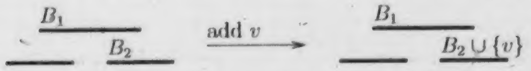
3. $k \geq 3$. By Lemma 6.13, v is fully adjacent to $B_2, \ldots, B_{k-1}$. Let $B_\alpha = F_R(B_1)$ and $B_\beta = F_L(B_k)$. As well, let b_α be some vertex in B_α, and b_β be some vertex in B_β. Observe that $B_1 \prec B_\alpha$ and $B_\beta \prec B_k$, otherwise, B_1 and B_k do not belong to the same component. Moreover, by definition, $F_L(B_\alpha) \preceq B_1$ and $B_k \preceq F_R(B_\beta)$. Furthermore, if $B_\alpha \prec B_\beta$, then there cannot be another block B for which $B_\alpha \prec B \prec B_\beta$ $[b(I_R(b_\alpha)) \neq b(b_\beta)]$. Otherwise, condition 3 of Lemma 6.13 is violated. Note that the algorithm LEFTCOMPONENTBLOCKSTRUCTURE would have detected this violation.

   We consider four cases.

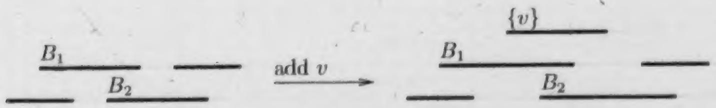
   (a) v is fully adjacent to B_k, and partially adjacent to B_1. In this case, $B_\alpha \prec B(v)$ as B_α is adjacent to B_1. Since B_α is adjacent to $F_R(B_\alpha)$, $B(v)$ is also adjacent to $F_R(B_\alpha)$, so we must ensure that $F_R(B_\alpha) \preceq B_k$ $[f_R(b_\alpha) \leq b(v_k)]$; otherwise, G' is not a proper interval graph.



(a) $F_L(B_2) \prec B_1$



(b) $F_L(B_2) = B_1$ and $F_R(B_2) = B_2$



(c) $F_L(B_2) = B_1$ and $B_2 \prec F_R(B_2)$

Figure 6.8: Adding the vertex v, where v is adjacent only to B_1 and B_2 and $F_L(B_1) \prec B_1$. Without loss of generality, we have assumed that v is fully adjacent to B_1

Now $B_1 \prec F_L(I_R(B_\alpha))$, therefore, $B_\alpha \prec B(v) \prec I_R(B_\alpha)$. Yet v is adjacent to B_k, so we must also ensure that $B_k \preceq F_R(I_R(B_\alpha))$ $[b(v_k) \leq f_R(I_R(b_\alpha))]$; otherwise, G' is not a proper interval graph. To create the new contig, we partition B_1 into $B_1 \setminus X \prec B_1 \cap X$ and insert the block $\{v\}$ immediately after B_α, as depicted Figure 6.9(a).

(b) v is fully adjacent to B_1, and partially adjacent to B_k. In this case, $B(v) \preceq B_\beta$ as B_k is adjacent to B_β. Since B_β is adjacent to $F_L(B_\beta)$, $B(v)$ is also adjacent to $F_L(B_\beta)$, so we must ensure that $B_1 \preceq F_L(B_\beta)$ $[b(v_1) \leq f_L(b_\beta)]$; otherwise, G' is not a proper interval graph.

Now $F_R(I_L(B_\beta)) \prec B_k$, therefore, $I_L(B_\beta) \prec B(v) \prec B_\beta$. Yet v is adjacent to B_1, so we must also ensure that $F_L(I_L(B_\beta)) \preceq B_1$ $[f_L(I_L(b_\beta)) \leq b(v_1)]$; otherwise, G' is not a proper interval graph.

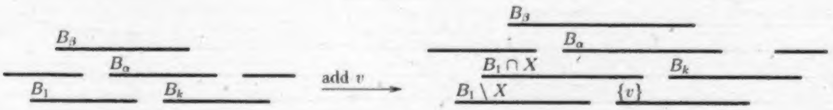
To create the new contig we partition B_k into $B_k \cap X \prec B_k \setminus X$, and insert the block $B_e = \{v\}$ immediately before B_β. This scenario is virtually identical to the case when v was fully adjacent to B_k and partially adjacent to B_1, and is depicted in Figure 6.9(b).

(c) v is partially adjacent to both B_1 and B_k. As we have seen, these conditions necessitate that $B_\alpha \prec B(v)$ and $B(v) \prec B_\beta$. As such, if $B_\beta \preceq B_\alpha$ $[b(b_\beta) \leq b(b_\alpha)]$, then G' cannot be a proper interval graph.

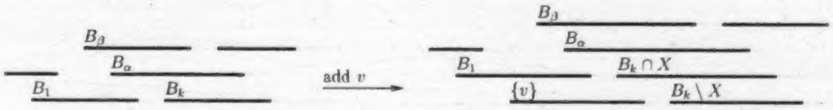
From the previous cases, we also require that $F_R(B_\alpha) \preceq B_k$ $[f_R(b_\alpha) \leq b(v_k)]$

and $B_1 \preceq F_L(B_\beta)$ $[b(v_1) \leq f_L(b_\beta)]$. However, given that v is partially adjacent to both B_1, and B_k, we actually have a slightly stronger requirement, namely, $F_R(B_\alpha) \prec B_k$ $[f_R(b_\alpha) < b(v_k)]$ and $B_1 \prec F_L(B_\beta)$ $[b(v_1) < f_L(b_\beta)]$. Providing $B_\alpha \prec B_\beta$, both of these conditions are satisfied.

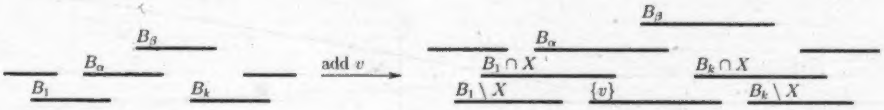
In essence, this scenario requires the "combination" of the two previous relabellings. That is, we partition B_1 into $B_1 \setminus X \prec B_1 \cap X$, we partition B_k into $B_k \cap X \prec B_k \setminus X$, and we insert the block $\{v\}$ between B_α and B_β. This scenario is depicted in Figure 6.9(c).


B_β
B_α
B_1
B_k
add v
B_β
B_α
$B_1 \cap X$
B_k
$B_1 \setminus X$
$\{v\}$

(a) v is fully adjacent to B_k and partially adjacent to B_1


B_β
B_α
B_1
B_k
add v
B_β
B_α
B_1
$B_k \cap X$
$\{v\}$
$B_k \setminus X$

(b) v is fully adjacent to B_1 and partially adjacent to B_k


B_β
B_α
B_1
B_k
add v
B_β
B_α
$B_1 \cap X$
$B_k \cap X$
$B_1 \setminus X$
$\{v\}$
$B_k \setminus X$

(c) v is partially adjacent to both B_1 and B_k

Figure 6.9: Adding the vertex v, where v is adjacent to B_1 through B_k $(k \geq 3)$

(d) v is fully adjacent to both B_1 and B_k. We consider three further cases.

  i. $B_\alpha \prec B_\beta$ $[b(b_\alpha) < b(b_\beta)]$. Using an earlier argument, we know that there cannot be a block B for which $B_\alpha \prec B \prec B_\beta$ $[b(I_R(b_\alpha)) \neq b(b_\beta)]$. Now if $B(v) \preceq B_\alpha$, then $B_\alpha \prec B_\beta$ gives that $B(v)$ is not adjacent to B_k, which is a contradiction. Similarly, if $B_\beta \preceq B(v)$, then $B_\alpha \prec B_\beta$ gives that $B(v)$ is not adjacent to B_1, which is a contradiction. Thereby, $B_\alpha \prec B(v) \prec B_\beta$, so we add the block $\{v\}$ between B_α and B_β, as shown in Figure 6.10.

  ii. $B_\alpha = B_\beta$ $[b(b_\alpha) = b(b_\beta)]$. We consider four cases.

    - $F_L(B_\alpha) = B_1$ $[f_L(b_\alpha) = b(v_1)]$ and $F_R(B_\alpha) = B_k$ $[f_R(b_\alpha) = b(v_k)]$. Since B_α has the same adjacency as v, we add v to B_α, as depicted in Figure 6.11(a).

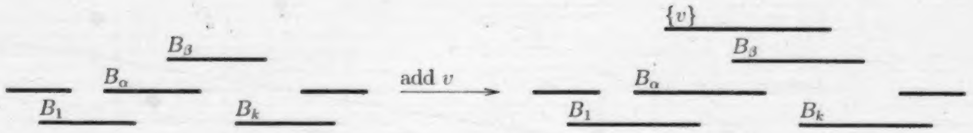
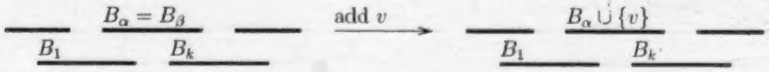
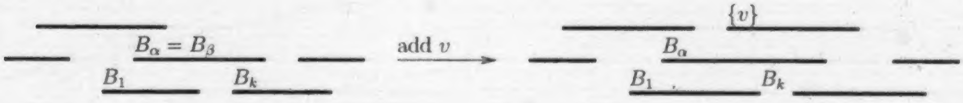

$\{v\}$
B_β
B_β
B_α
add v
B_α
B_1
B_k
B_1
B_k

Figure 6.10: Adding the vertex v, where v is fully adjacent to B_1 through B_k ($k \geq 3$), and $B_\alpha = F_R(B_1) \prec F_L(B_k) = B_\beta$

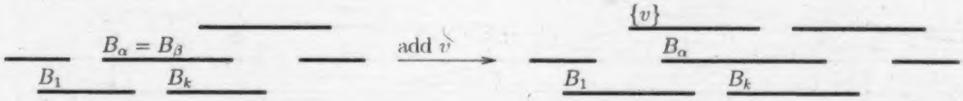
- $F_L(B_\alpha) \prec B_1$ $[f_L(b_\alpha) < b(v_1)]$ and $F_R(B_\alpha) = B_k$ $[f_R(b_\alpha) = b(v_k)]$. Since $F_L(B_\alpha) \prec B_1$, $B_\alpha \prec B(v)$. Furthermore, observe that $B_1 \prec F_L(I_R(B_\alpha))$, so $B(v) \prec I_R(B_\alpha)$. As such, we must insert the block $\{v\}$ immediately after B_α, as shown in Figure 6.11(b).
- $F_L(B_\alpha) = B_1$ $[f_L(b_\alpha) = b(v_1)]$ and $B_k \prec F_R(B_\alpha)$ $[b(v_k) < f_R(b_\alpha)]$. An argument similar to the preceding one gives that the block $\{v\}$ must be inserted immediately before B_α, as shown in Figure 6.11(c).
- $F_L(B_\alpha) \prec B_1$ $[f_L(b_\alpha) < b(v_1)]$ and $B_k \prec F_R(B_\alpha)$ $[b(v_k) < f_R(b_\alpha)]$. From the previous cases, we see that $B_\alpha \prec B(v)$ and $B(v) \prec B_\alpha$, which is a contradiction. Thereby, G' is not a proper interval graph.


$B_\alpha = B_\beta$
add v
$B_\alpha \cup \{v\}$
B_1
B_k
B_1
B_k

(a) $F_L(B_\alpha) = B_1$, and $F_R(B_\alpha) = B_k$


$\{v\}$
$B_\alpha = B_\beta$
add v
B_α
B_1
B_k
B_1
B_k

(b) $F_L(B_\alpha) \prec B_1$, and $F_R(B_\alpha) = B_k$


$\{v\}$
$B_\alpha = B_\beta$
add v
B_α
B_1
B_k
B_1
B_k

(c) $F_L(B_\alpha) = B_1$, and $B_k \prec F_R(B_\alpha)$

Figure 6.11: Adding the vertex v, where v is fully adjacent to B_1 through B_k ($k \geq 3$), and $B_\alpha = F_R(B_1) = F_L(B_k) = B_\beta$

iii. $B_\beta \prec B_\alpha$ $[b(b_\beta) < b(b_\alpha)]$. By definition, $F_L(B_\alpha) \preceq B_1$ and $B_k \preceq F_R(B_\beta)$, therefore, $F_L(B_\beta) \preceq B_1$ and $B_k \preceq F_R(B_\alpha)$. We consider four cases.

- $F_L(B_\beta) = B_1$ $[f_L(b_\beta) = b(v_1)]$ and $F_R(B_\alpha) = B_k$ $[f_R(b_\alpha) = b(v_k)]$. Since $F_L(B_\beta) = B_1$, $F_L(B_\beta) \preceq F_L(B_\alpha)$ gives $B_1 \preceq F_L(B_\alpha)$, so $F_L(B_\alpha) =$

$F_L(B_\beta) = B_1$. Similarly, $F_R(B_\beta) = F_R(B_\alpha) = B_k$, so $B_\alpha = B_\beta$. This contradiction tells us that this case cannot occur.

- $F_L(B_\beta) \prec B_1$ $[f_L(b_\beta) < b(v_1)]$ and $F_R(B_\alpha) = B_k$ $[f_R(b_\alpha) = b(v_k)]$. Since $F_L(B_\beta) \prec B_1$, $B_\beta \prec B(v)$. As such, $F_R(B_\beta) \preceq B_k$ $[f_R(b_\beta) \leq b(v_k)]$, otherwise, G' is not a proper interval graph.

  Now if $F_L(B_\alpha) = B_1$ $[f_L(b_\alpha) = b(v_1)]$, then B_α has the same adjacency as v so we add v to B_α, as illustrated in Figure 6.12(a). Otherwise, if $F_L(B_\alpha) \prec B_1$ $[f_L(b_\alpha) < b(v_1)]$, we must insert the block $\{v\}$ immediately after B_α, as illustrated in Figure 6.12(b), because $B_1 \prec F_L(I_R(B_\alpha))$.

- $F_L(B_\beta) = B_1$ $[f_L(b_\beta) = b(v_1)]$ and $B_k \prec F_R(B_\alpha)$ $[b(v_k) < f_R(b_\alpha)]$. This case is virtually identical to the previous one. As such, we we must check that $B_1 \preceq F_L(B_\alpha)$ $[B(v_1) \leq f_L(b_\alpha)]$.

  If $F_R(B_\beta) = B_k$ $[f_R(b_\beta) = b(v_k)]$, then we add v to B_β, as shown in Figure 6.12(c). Otherwise, if $B_k \prec F_R(B_\beta)$ $[b(v_k) < f_R(b_\beta)]$, we must insert the block $\{v\}$ immediately before B_β, as illustrated in Figure 6.12(d).

- $F_L(B_\beta) \prec B_1$ $[f_L(b_\beta) < b(v_1)]$ and $B_k \prec F_R(B_\alpha)$ $[b(v_k) < f_R(b_\alpha)]$. Since $B_k \prec F_R(B_\alpha)$, $B(v) \prec B_\alpha$. Now if $F_L(B_\alpha) \prec B_1$ $[f_L(b_\alpha) < b(v_1)]$, then $B_\alpha \prec B(v)$, which is a contradiction that tells us that G' is not a proper interval graph. Therefore, $F_L(B_\alpha) = B_1$. Similarly, $F_R(B_\beta) = B_k$, and we must verify that $F_R(B_\beta) \preceq B_k$ $[f_R(b_\beta) \leq b(v_k)]$.

  Now consider $F_R(I_L(B_1))$. Since $F_L(F_R(I_L(B_1))) \preceq (I_L(B_1)) \prec B_1$, $F_R(I_L(B_1)) \prec B(v)$. Similarly, $B(v) \prec F_L(I_R(B_k))$. As such, we must check the condition $F_R(I_L(B_1)) \preceq F_L(I_R(B_k))$ $[f_R(I_L(v_1)) \leq f_L(I_R(v_k))]$. Since $F_L(B_\beta) \prec B_1$, we also know that $B_\beta \preceq F_R(I_L(B_1))$; similarly, $F_L(I_R(B_k)) \preceq B_\alpha$.

  If there exists a block B such that $F_R(I_L(B_1)) \prec B \prec F_L(I_R(B_k))$, then $B_1 \preceq F_L(B)$ and $F_R(B) \preceq B_k$. Yet $B_\beta \prec B$, where $F_R(B_\beta) = B_k$, therefore, $F_R(B) = B_k$. Similarly, $B_1 = F_L(B)$, so we must add v to B as shown in Figure 6.12(e). Note that we have clearly defined $F_L(B)$ and $F_R(B)$, therefore, there is only one such block B.

  If there does not exist a block B such that $F_R(I_L(B_1)) \prec B \prec F_L(I_R(B_k))$, then we must add the block $\{v\}$ between $F_R(I_L(B_1))$ and $F_L(I_R(B_k))$. This scenario is also shown in Figure 6.12(e).

Now let us consider when the members of X belong to two distinct components. From condition 5 of Lemma 6.13 we know that each segment must contain an end block to which v is fully adjacent. We add v by merging the two contigs and placing the block $\{v\}$ between. Let the contigs containing the two segments be $\Phi = B_1 \prec \ldots \prec B_k$ and $\Psi = B'_1 \prec \ldots \prec$

B'_l, where, without loss of generality, $\Phi \prec \Psi$. Note that we can follow I_L and I_R pointers, as necessary, to determine the end blocks of Φ and Ψ. If B_k and B'_1 are the end blocks to which v is fully adjacent, then the new merged contig will be $\Phi \prec \{v\} \prec \Psi$, however, we note that it may also be necessary to split some of the blocks in the merged contig. Construction of this new merged contig requires us to move Φ over to Ψ.

Let B_i be the leftmost block in Φ to which v is adjacent, and let B'_j be the rightmost block in Ψ to which v is adjacent. If v is partially adjacent to B_i, then we must partition B_i into $B_i \setminus X \prec B_i \cap X$. Similarly, if v is partially adjacent to B'_j, then we must partition B'_j into $B'_j \cap X \prec B'_j \setminus X$. The change in the straight enumeration is illustrated in Figure 6.13 which considers when the end blocks to which v is fully adjacent are B_k and B'_1, with v fully adjacent to B'_j, but not B_i.

Similar merges are done for the other scenarios in which the end blocks fully adjacent to v are B_k and B'_l, B_1 and B'_1, and B_1 and B'_l. In the case where the end blocks are fully adjacent with v are B_k and B'_l, the merged contig is $\Psi \prec \{v\} \prec \Phi^R$; this merge amounts to a move and "flip" of Φ.

As we have seen, the algorithm LeftComponentBlockStructure allows to determine a great deal of information about the contigs containing members of X. The correctness, at least the block level, of our algorithm is due to the careful consideration of the exhaustive cases which can be identified according to information obtained from LeftComponentBlockStructure, as well as specific vertex-level information. Again, ensuring that the algorithm does what we want it to do, in terms of pointers and vertex labels, is a matter of verification within each individual case. Recall that many of the vertex-level instructions are presented in Appendix C.

Deleting an edge

Let uv be the edge to be deleted, where X_u and X_v denote the neighbourhoods of u and v in G, respectively. As well, let B_i and B_j be the blocks containing u and v, respectively, in the contig $B_1 \prec \ldots \prec B_k$ of the component C containing uv. Without loss of generality, let $1 \leq i \leq j \leq k$.

The following theorem addresses the case where $i = j$.

Lemma 6.14 *[27] Let u and v be two adjacent vertices in a proper interval graph G. If $N[u] = N[v]$, then $G - uv$ is a proper interval graph if and only if the component containing u and v is a clique.*

Consequently, if $i = j$ $[b(u) = b(v)]$, then $i = j = k = 1$ $[f_L(v) = f_R(v)]$; otherwise, G' is not a proper interval graph. In this case, we partition the contig B_1 to create a new contig $\{u\} \prec B_1 \setminus \{u, v\} \prec \{v\}$, as depicted in Figure 6.14(a).

On the other hand, if $i \neq j$, then the relabelling is slightly more complicated. In this case, the following lemma applies.

Lemma 6.15 *[27] Let u and v be adjacent vertices of a proper interval graph G. As well, let $B_1 \prec \ldots \prec B_k$ be a contig of G, such that $u \in B_i$ and $v \in B_j$, for some $1 \leq i < j \leq k$. The graph $G - uv$ is a proper interval graph if and only if $F_R(B_i) = B_j$ and $F_L(B_j) = B_i$.*

Consequently, if $i \neq j$, then $F_R(B_i) = B_j$ $[f_R(u) = b(v)]$ and $F_L(B_j) = B_i$ $[f_L(v) = b(u)]$; otherwise, G' is not a proper interval graph. Observe that if $1 < i$ $[f_L(u) \neq b(u)]$, $B_j = \{v\}$ $[nx(v) = v]$, $F_L(B_{i-1}) = F_L(B_i)$ $[f_L(I_L(u)) = f_L(u)]$, and $F_R(B_{i-1}) = B_{j-1}$ $[f_R(I_L(u)) = b(I_L(v))]$, then we must move u into B_{i-1}. Similarly, if $j < k$, $B_i = \{u\}$, $F_R(B_{j+1}) = F_R(B_j)$, and $F_L(B_{j+1}) = B_{i+1}$, then we must move v into B_{j+1}.

Exactly how the labelling is changed depends on whether u is moved into B_{i-1}, v is moved into B_{j+1}, $B_i = \{u\}$, and $B_j = \{v\}$. We consider each case, with respect to u, separately, noting that the same considerations must also be given for v.

- If u is to be moved into B_{i-1}, then the straight enumeration changes as shown in Figure 6.14(b).
- If u was not moved into B_{i-1} and $B_i = \{u\}$, then the straight enumeration changes as shown in Figure 6.14(c).
- If u was not moved into B_{i-1} and B_i contains vertices other than u, then we must partition B_i into $\{u\} \prec B_i \setminus \{u\}$ (in the case of v, we would partition B_j into $B_j \setminus \{v\} \prec \{v\}$). This scenario is depicted in Figure 6.14(d).

The correctness, at least the block level, of our algorithm is due to the careful consideration of the exhaustive cases. Again, ensuring that the algorithm does what we want it to do, in terms of pointers and vertex labels, is a matter of verification.

Adding an edge

Let uv be the edge to be added, where X_u and X_v denote the neighbourhoods of u and v in G, respectively. The following lemmas characterize when G' is a proper interval graph.

Lemma 6.16 *[27] If u and v are in distinct components of a proper interval graph G, then $G + uv$ is a proper interval graph if and only if u and v are end vertices in a straight enumeration of G.*

Lemma 6.17 *[27] Let u and v be non-adjacent vertices belonging to the same component of a proper interval graph G. As well, let $B_1 \prec \ldots \prec B_k$ be a contig of that component, where $u \in B_i$ and $v \in B_j$, for some $1 \leq i < j \leq k$. The graph $G + uv$ is a proper interval graph if and only if $F_R(B_i) = B_{j-1}$ and $F_L(B_j) = B_{i+1}$.*

Without loss of generality, let us assume that $b(u) < b(v)$. While considering the addition of a vertex, we saw that, by traversing F_R pointers beginning at u, we can determine, in $O(n)$ time, whether u and v belong to the same component. If u and v belong to the same component, the conditions $F_R(B_i) = B_{j-1}$ [$f_R(u) = b(I_L(v))$] and $F_L(B_j) = B_{i+1}$ [$f_L(v) = b(I_R(v))$] must be satisfied; otherwise, G' is not a proper interval graph. On the other hand, if u and v do not belong to the same component, u and v must be end vertices [$f_L(u) = b(u)$ or $f_R(u) = b(u)$, and $f_L(v) = b(v)$ or $f_R(v) = b(v)$]; otherwise, G' is not a proper interval graph.

Having determined whether G' is a proper interval graph, we consider the following cases.

1. The vertices u and v belong to distinct components. In this case we will need to know information about all the blocks in the components containing u and v. Specifically, we determine all the blocks by following F_L and F_R pointers, keeping a reference vertex v_i from each block B_i. Gathering this information can take as much as $\Theta(n)$ time.

   Let $\Phi = B_1 \prec \ldots \prec B_k$ be the contig of the component containing u, and let $\Psi = B'_1 \prec \ldots \prec B'_l$ be the contig of the component containing v. If $u \in B_k$ and $v \in B'_1$, then the new merged contig will be $\Phi \prec \Psi$; however, we note that it may also be necessary to split some of the blocks in the merged contig. Specifically, if $B_k \neq \{u\}$ [$nx(u) \neq u$], then we must partition B_k into $B_k \setminus \{u\} \prec \{u\}$; similarly, if $B'_1 \neq \{v\}$ [$nx(v) \neq v$], then we must partition B'_1 into $\{v\} \prec B'_1 \setminus \{v\}$. Similar merges and splits are done for the other scenarios in which $u \in B_k$ and $v \in B'_l$, $u \in B_1$ and $v \in B'_1$, and $u \in B_1$ and $v \in B'_l$.

   The change in the straight enumeration is illustrated in Figure 6.15(a) which considers when $u \in B_k$ and $v \in B'_1$, with $B'_1 = \{v\}$ but not $B_k \neq \{u\}$.

2. The vertices u and v belong to the same component. As per the hypothesis of Lemma 6.17, let $B_1 \prec \ldots \prec B_k$ be the contig of the component containing u and v, where $u \in B_i$ and $v \in B_j$, for some $1 \leq i < j \leq k$. We consider two further cases.

   (a) B_i and B_j are end blocks [$f_L(u) = b(u)$ and $f_R(v) = b(v)$]. In this case, $X_u = X_v$. By Lemma 6.7 (umbrella property), the contig contains three blocks, namely, $\{u\} \prec X_u \prec \{v\}$. The new component will consist of a single block, formed by merging the three blocks into one new block, as shown in Figure 6.15(b).

   (b) At least one of B_i and B_j is not an end block [$f_L(u) \neq b(u)$ or $f_R(v) \neq b(v)$]. In this case, $X_u \neq X_v$. If $B_i = \{u\}$ [$nx(u) = u$], $F_R(B_{j-1}) = F_R(B_j)$ [$f_R(I_L(v)) = f_R(v)$] and $F_L(B_{j-1}) = B_i$ [$f_L(I_L(v)) = b(u)$], then we move v from B_j into B_{j-1}. Similarly, if $B_j = \{v\}$ [$nx(v) = v$], $F_L(B_{i+1}) = F_L(B_i)$ [$f_L(I_R(u)) = f_L(u)$]

and $F_R(B_{i+1}) = B_j$ $[f_R(I_R(u)) = b(v)]$, then we move u from B_i into B_{i+1}. This moving of u and v is virtually identical to one of the cases discussed when considering the deletion of a edge.

If u was not moved, and B_i contains vertices other that u, then we partition B_i into $B_i \setminus \{u\} \prec \{u\}$. Similarly, if v was not moved, and B_j contains vertices other that v, then we partition B_j into $\{v\} \prec B_j \setminus \{v\}$.

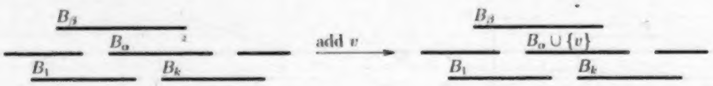
Exactly how the labelling is changed depends on whether u is moved into B_{i+1}, v is moved into B_{j-1}, $B_i = \{u\}$, and $B_j = \{v\}$. We consider each case, with respect to u, separately, noting that the same considerations must also be given for v.

- If u is to be moved into B_{i+1}, then the straight enumeration changes as shown in Figure 6.15(c).
- If u was not moved into B_{i+1} and $B_i = \{u\}$, then the straight enumeration changes as shown in Figure 6.15(d).
- If u was not moved into B_{i+1} and B_i contains vertices other than u, then we must partition B_i into $B_i \setminus \{u\} \prec \{u\}$ (in the case of v, we would partition B_j into $\{v\} \prec B_j \setminus \{v\}$). This scenario is depicted in Figure 6.15(e).

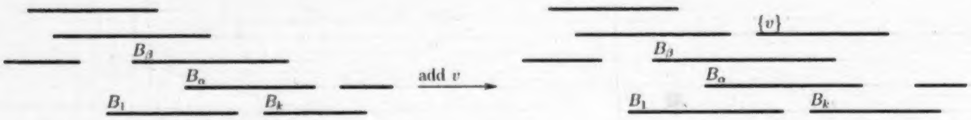
The correctness, at least the block level, of our algorithm is due to the careful consideration of the exhaustive cases. Again, ensuring that the algorithm does what we want it to do, in terms of pointers and vertex labels, is a matter of verification.

6.2 Summary

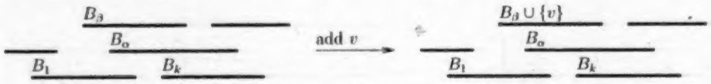
In this chapter, we apply a distributed pointer technique, along with the circular doubly linked list technique seen in Chapters 4 and 5, in order to develop error-detecting dynamic adjacency labelling schemes for proper interval graphs. Our dynamic scheme, which is largely based on a centralized scheme of Hell, Shamir, and Sharan [27], uses $O(\log n)$ bit labels and handles all operations in $O(n)$ time.


B_β
B_α
B_1
B_k
add v
B_β
$B_\alpha \cup \{v\}$
B_1
B_k

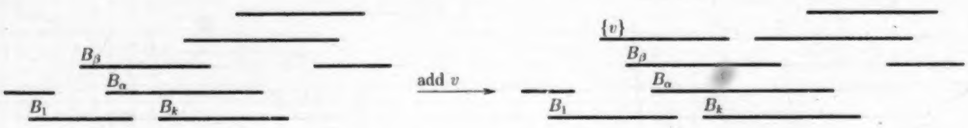
(a) $F_L(B_\beta) \prec B_1$, $F_R(B_\alpha) = B_k$, and $F_L(B_\alpha) = B_1$


B_β
B_α
B_1
B_k
add v
$\{v\}$
B_β
B_α
B_1
B_k

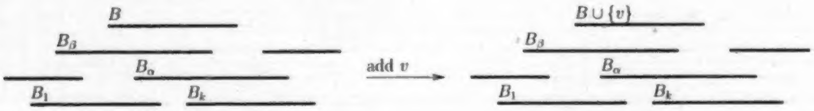
(b) $F_L(B_\beta) \prec B_1$, $F_R(B_\alpha) = B_k$, and $F_L(B_\alpha) \prec B_1$


B_β
B_α
B_1
B_k
add v
$B_\beta \cup \{v\}$
B_α
B_1
B_k

(c) $F_L(B_\beta) = B_1$, $B_k \prec F_R(B_\alpha)$, and $F_R(B_\beta) = B_k$


B_β
B_α
B_1
B_k
add v
$\{v\}$
B_β
B_α
B_1
B_k

(d) $F_L(B_\beta) = B_1$, $B_k \prec F_R(B_\alpha)$, and $B_k \prec F_R(B_\beta)$


B
B_β
B_α
B_1
B_k
add v
$B \cup \{v\}$
B_β
B_α
B_1
B_k

(e) $F_L(B_\beta) \prec B_1$ and $B_k \prec F_R(B_\alpha)$. Note that the block B may be empty

Figure 6.12: Adding the vertex v, where v is fully adjacent to B_1 through B_k ($k \geq 3$), and $B_\beta = F_L(B_k) \prec F_R(B_1) \doteq B_\alpha$

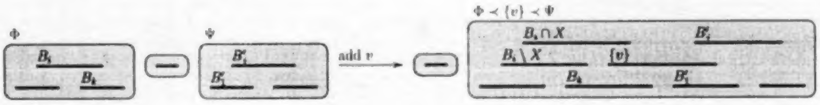
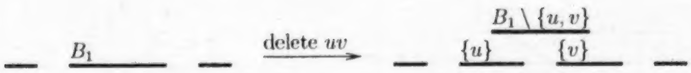
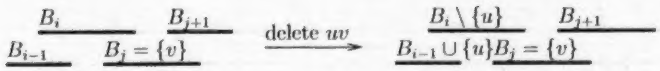

Φ
B_i
B_k
Ψ
B'_1
B'_j
add v
$\Phi \prec \{v\} \prec \Psi$
$B_i \cap X$
B'_j
$B_i \setminus X$
$\{v\}$
B_k
B'_1

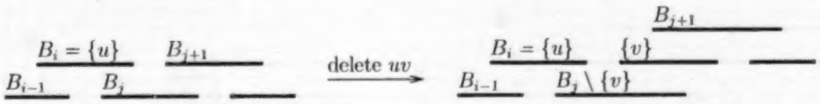
Figure 6.13: Adding the vertex v, where the neighbours of v span more than one component. In this case, v is fully adjacent with $B_{i+1}, \ldots, B_k$ and $B'_1, \ldots, B'_j$, and partially adjacent with B_i


B_1
delete uv
$B_1 \setminus \{u,v\}$
$\{u\}$
$\{v\}$

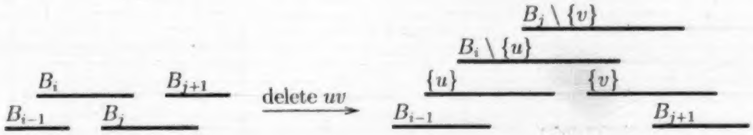
(a) $N[u] = N[v]$


B_i
B_{j+1}
B_{i-1}
$B_j = \{v\}$
delete uv
$B_i \setminus \{u\}$
B_{j+1}
$B_{i-1} \cup \{u\}$
$B_j = \{v\}$

(b) $N[u] \neq N[v]$, u is moved into B_{i-1}, and v is not moved into B_{j+1}

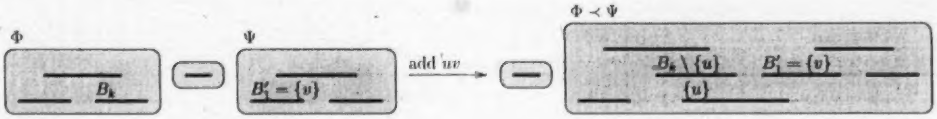

$B_i = \{u\}$
B_{j+1}
B_{i-1}
B_j
delete uv
B_{j+1}
$B_i = \{u\}$
$\{v\}$
B_{i-1}
$B_j \setminus \{v\}$

(c) $N[u] \neq N[v]$, u is not moved into B_{i-1}, and $B_i = \{u\}$


B_i
B_{j+1}
B_{i-1}
B_j
delete uv
$B_j \setminus \{v\}$
$B_i \setminus \{u\}$
$\{u\}$
$\{v\}$
B_{i-1}
B_{j+1}

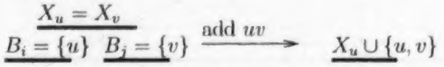
(d) $N[u] \neq N[v]$, u is not moved into B_{i-1}, and $B_i \neq \{u\}$

Figure 6.14: Deleting the edge uv



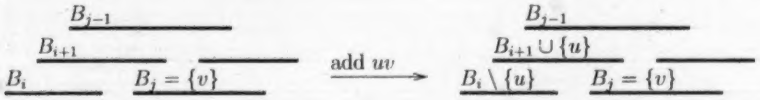
Φ
Ψ
Φ ≺ Ψ
B_k
$B'_1 = \{v\}$
add uv
$B_k \setminus \{u\}$
$B'_1 = \{v\}$
$\{u\}$

(a) The vertices u and v belong to distinct components. In this case, $u \in B_k$, $v \in B'_1$, $B_k \neq \{u\}$, and $B'_1 = \{v\}$



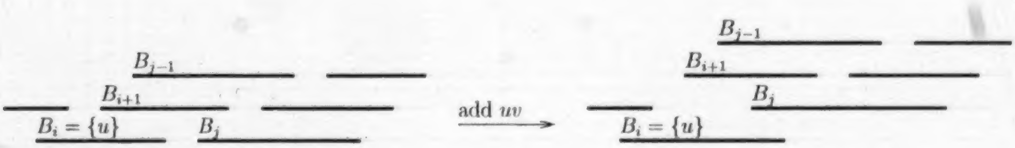
$X_u = X_v$
$B_i = \{u\}$
$B_j = \{v\}$
add uv
$X_u \cup \{u, v\}$

(b) The vertices u and v belong to the same component. In this case, both B_i and B_j are end blocks



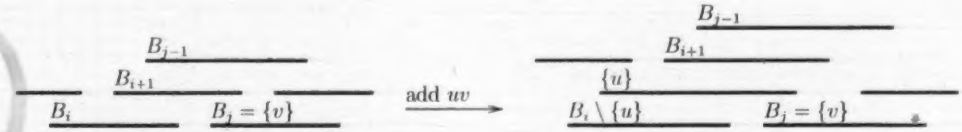
B_{j-1}
B_{i+1}
B_i
$B_j = \{v\}$
add uv
B_{j-1}
$B_{i+1} \cup \{u\}$
$B_i \setminus \{u\}$
$B_j = \{v\}$

(c) The vertices u and v belong to the same component. In this case, at least one of B_i and B_j is an end block, u is moved into B_{i+1}, and v is not moved into B_{j-1}



B_{j-1}
B_{i+1}
$B_i = \{u\}$
B_j
add uv
B_{j-1}
B_{i+1}
B_j
$B_i = \{u\}$

(d) The vertices u and v belong to the same component. In this case, at least one of B_i and B_j is an end block, u is not moved into B_{i+1}, and $B_i = \{u\}$



B_{j-1}
B_{i+1}
B_i
$B_j = \{v\}$
add uv
B_{j-1}
B_{i+1}
$\{u\}$
$B_i \setminus \{u\}$
$B_j = \{v\}$

(e) The vertices u and v belong to the same component. In this case, at least one of B_i and B_j is an end block, u is not moved into B_{i+1}, and $B_i \neq \{u\}$

Figure 6.15: Adding the edge uv

Chapter 7

Conclusion

In order to increase the applicability of informative labelling schemes to real world problems in which the underlying topology is constantly changing, we have formally defined the concept of a dynamic informative labelling scheme. There have been earlier publications on this subject, but these works have been based exclusively on our intuitive understanding of how static problems are dynamized. While presenting this definition, we introduced the concept of error-detection, in which the relabeller recognizes when the modified graph is no longer a member of the family under consideration. Additionally, we demonstrated the connection between error-detection and the graph recognition problem, and identified and discussed the qualities that make a good dynamic scheme.

The latter half of our work was dedicated to the development of error-detecting dynamic adjacency labelling schemes for four classes of graphs. Common to the development of dynamic schemes for all these classes was the use of a technique that employed circular doubly linked lists to encode information about graph substructures at the vertex level. Moreover, for one of the classes we developed a technique to distribute pointers. Each of the dynamic adjacency labelling schemes that we developed was fully dynamic, that is, the allowed graph operations were the addition or deletion of a vertex (along with its incident edges), and the addition or deletion of an edge.

In the case of line graphs, our dynamic scheme used $O(\log n)$ bit labels and updates could be performed in $O(e)$ time, where e was the number of edges added to, or deleted from, the line graph. In developing this dynamic scheme, we introduced a new concept known as partition isomorphism, and developed theory regarding the types of line graphs that can be changed to produce new line graphs.

In the cases of r-minoes, defined by Metelsky and Tyshkevich [44] as the class of graphs with no vertex in more than r maximal cliques, our dynamic scheme used $O(r \log n)$ bit labels. Edge addition and deletion were handled in $O(r^2\mathbf{D})$ time, vertex addition in $O(r^2e^2)$ time, and vertex deletion in $O(r^2e)$ time, where $\mathbf{D}$ was the maximum degree of the vertices in the original graph and e was the number of edges added to, or deleted from, the original graph.

In the case of r-bics, a new class which we defined to be the graphs with no vertex in more than r maximal bicliques, our dynamic schemes used $O(r \log n)$ bit labels. Edge addition and deletion, as well as vertex deletion, were handled in $O(r^2\mathbf{B})$ time, and vertex addition in $O(r^2 n\mathbf{B})$ time, where **B** was the size of the largest biclique in the original graph.

Finally, in the case of proper interval graphs, our dynamic scheme used $O(\log n)$ bit labels and handled all operations in $O(n)$ time.

Our work on dynamic informative labelling schemes leaves several open questions.

1. What general mechanisms can be developed for creating dynamic schemes from static schemes (besides recreating the graph and running the marker each time the graph is changed)? The work of Korman, Peleg, and Rodeh [39], offers such a technique for the dynamic weighted trees, when considering any function f that satisfies the following properties.

   - For any two vertices u and v, $f(u, v)$ depends entirely on the path between them.
   - For any three vertices u, v, and w, where w is on the path between u and v, $f(u, v)$ can be calculated in polynomial time from $f(u, w)$ and $f(w, v)$.

   Such functions include routing, distance, separation level, and flow. Instead of fixing the graph class and developing mechanisms for different functions, can we fix the function and develop mechanisms for different graph classes?

2. Is there a dynamic adjacency labelling scheme for proper interval graphs that uses $o(\log n)$ bit labels. We know that there are $2^{\Theta(n)}$ proper interval graphs on n vertices [22], so there could be a dynamic scheme with labels that use $O(1)$ bits. The existence of such a dynamic scheme would imply the the existence of an adjacency labelling scheme that uses $O(1)$ bit labels.

3. Is there a dynamic adjacency labelling scheme for proper interval graphs that uses $\Theta(\log n)$ bit labels, yet allows relabelling in $o(n)$ time? Unfortunately, the dynamic scheme presented for proper interval graphs in Chapter 6 is hampered by the fact that we must maintain the straight enumeration, which necessitates using as much as $\Theta(n)$ time for each graph operation.

4. Is there a dynamic adjacency labelling scheme for interval graphs that uses $O(\log n)$ bit labels. Interval graphs, which are not all that different from proper interval graphs, exhibit an adjacency labelling scheme that uses $O(\log n)$ bit labels [45]. It makes sense that the dynamic scheme for proper interval graphs presented in Chapter 6 might be extended to give a dynamic scheme for interval graphs.

5. Can we devise dynamic informative labelling schemes for functions other than adjacency, over and above the work that has already been done on trees [37, 39]? The class of trees, although relevant to many applications, is typically the easiest family on which to consider a graph theoretical problem. Are we able to devise such dynamic schemes for classes of size $2^{\omega(n)}$?

Bibliography

[1] S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (Washington, D.C., USA)*, pages 547–556, 2001.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, 1974.

[3] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Identifying nearest common ancestors in a distributed environment. IT-C Technical Report Series 2001-6, The IT University of Copenhagen, 2001.

[4] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new distributed algorithm. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (Winnipeg, Canada)*, pages 258–264, 2002.

[5] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory of Computing Systems*, Online first:OF1–OF16, 2004.

[6] S. Alstrup and T. Rauhe. Small induced-universal graphs and compact implicit graph representations. In *43^{rd} Annual Symposium on Foundations of Computer Science (Vancouver, Canada)*, pages 53–62. IEEE, 2002.

[7] R. B. Borie, R. G. Parker, and C. A. Tovey. Recursively constructed graphs. In J. L. Gross and J. Yellen, editors, *Handbook of Graph Theory*, pages 99–108. CRC Press, New York, 2003.

[8] A. Branstädt, V. B. Le, and J. P. Spinrad. *Graph Classes: A Survey.* SIAM Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia, 1999.

[9] R. C. Brigham. Bandwidth. In J. L. Gross and J. Yellen, editors, *Handbook of Graph Theory*, pages 922–944. CRC Press, New York, 2003.

[10] G. S. Brodal and R. Fagerberg. Dynamic representation of sparse graphs. In *Algorithms and Data Structures, Proceedings of the 6^{th} International Workshop (Vancouver, Canada)*, volume 1663 of *Lecture Notes in Computer Science*, pages 342–351. Springer-Verlag, 1999.

[11] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. In *Proceedings of the 21^{st} ACM Symposium on Principles of Database Systems (Madison, USA)*, pages 271–281. ACM, 2002.

[12] D. G. Corneil, H. Kim, S. Natarajan, S. Olariu, and A. P. Sprague. Simple linear time recognition of unit interval graphs. *Information Processing Letters*, 55:99–104, 1995.

[13] B. Courcelle and R. Vanicat. Query efficient implementation of graphs of bounded clique-width. *Discrete Applied Mathematics*, 131:129–150, 2003.

[14] X. Deng, P. Hell, and J. Huang. Linear-time representation algorithms for proper circular-arc graphs and proper interval graphs. *SIAM Journal on Computing*, 25:390–403, 1996.

[15] A. C. Driskell, C. Ané, J. G. Burleigh, M. M. McMahon, B. C. O'Meara, and M. J. Sanderson. Prospects for building the tree of life from large sequence databases. *Science*, 306:1172–1174, 2004.

[16] F. Fich. CSC 2429 - Advaced Data Structures: Lecture 1, 2000. www.cs.toronto.edu/~fich/DScourse/lecture1.ps.

[17] C. M. Fiduccia, E. R. Scheinerman, A. Trenk, and J. S. Zito. Dot product representations of graphs. *Discrete Mathematics*, 181:113–138, 1998.

[18] H. N. Gabow and H. H. Westermann. Forests, frames, and games: Algorithms for matroid sums and applications. *Algorithmica*, 7:465–497, 1992.

[19] C. Gavoille, M. Katz, N.A. Katz, C. Paul, and D. Peleg. Approximate distance labeling schemes. In *Proceedings of the 9^{th} European Symposium on Algorithms*, volume 2161 of *Lecture Notes in Computer Science*, pages 476–487. Springer-Verlag, 2001.

[20] C. Gavoille and C. Paul. Small universal distance matrices. Technical Report RR-1263-01, Laboratoire Bordelais de Recherche en Informatique, 2001.

[21] C. Gavoille and C. Paul. Distance labeling scheme and split decomposition. *Discrete Mathematics*, 273:115–130, 2003.

[22] C. Gavoille and C. Paul. Optimal distance labeling for interval and circular-arc graphs. In *Proceedings of the 11^{th} European Symposium on Algorithms*, volume 2832 of *Lecture Notes in Computer Science*, pages 254–265. Springer-Verlag, 2003.

[23] C. Gavoille and D. Peleg. Compact and localized distributed data structures. *Journal of Distributed Computing*, 16:111–120, 2003.

[24] C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance labeling in graphs. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (Washington, D.C., USA)*, pages 210–219. ACM, 2001.

[25] P. Hanlon. Counting interval graphs. *Transactions of the American Mathematical Society*, 272:383–426, 1982.

[26] F. Harary and E.M. Palmer. *Graphical Enumeration.* Academic Press, New York, 1973.

[27] P. Hell, R. Shamir, and R. Sharan. A fully dynamic algorithm for recognizing and representing proper interval graphs. *SIAM Journal on Computing*, 31(1):289–305, 2001.

[28] Z. Jackowski. A new characterization of proper interval graphs. *Discrete Mathematics*, 105:103–109, 1992.

[29] T. Jiang, M. Li, and B. Ravikumar. Basic notions in computational complexity. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 24. CRC Press, New York, 1998.

[30] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27:119–123, 1988.

[31] S. Kannan, M. Naor, and S. Rudich. Implicit representation of graphs. *SIAM Journal on Discrete Mathematics*, 5(4):596–603, 1992.

[32] H. Kaplan and T. Milo. Short and simple labels for small distances and other functions. In *Algorithms and Data Structures, Proceedings of the 7^{th} International Workshop (Providence, USA)*, volume 2125 of *Lecture Notes in Computer Science*, pages 246–257. Springer-Verlag, 2001.

[33] M. Katz, N. A. Katz, A. Korman, and D. Peleg. Labeling schemes for flow and connectivity. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (San Francisco, USA)*, pages 927–936. ACM, 2002.

[34] M. Katz, N. A. Katz, and D. Peleg. Distance labeling schemes for well-separated graph classes. Technical Report TRMCS99-26, The Weizmann Institute of Science, 1999.

[35] M. Katz, N. A. Katz, and D. Peleg. Distance labeling schemes for well-separeated graph classes. In *Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science (Lille, France)*, volume 1770 of *Lecture Notes in Computer Science*, pages 516–528. Springer-Verlag, 2000.

[36] T. Kloks, D. Kratsch, and H. Müller. Dominoes. In *Graph Theoretic Concepts in Computer Science, Proceedings of the 20th International Workshop (Herrsching, Germany)*, volume 903 of *Lecture Notes in Computer Science*, pages 106–120. Springer-Verlag, 1995.

[37] A. Korman and D. Peleg. Labeling schemes for weighted dynamic trees. In *Automata, Languages and Programming, Proceedings of the 30th International Colloquium (Eindhoven, The Netherlands)*, volume 2719 of *Lecture Notes in Computer Science*, pages 369–383. Springer-Verlag, 2003.

[38] A. Korman, D. Peleg, and Y. Rodeh. Labeling schemes for dynamic tree networks. In *Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science (Antibes - Juan les Pins, France)*, volume 2285 of *Lecture Notes in Computer Science*, pages 76–87. Springer-Verlag, 2002.

[39] A. Korman, D. Peleg, and Y. Rodeh. Labeling schemes for dynamic tree networks. *Theory of Computing Systems*, 37:49–75, 2004.

[40] P. G. H. Lehot. An optimal algorithm to detect a line graph and output its root graph. *Journal of the Association of Computing Machines*, 21(4):569–575, 1974.

[41] P. J. Looges and S. Olariu. Optimal greedy algorithms for indifference graphs. *Computers and Mathematics with Applications*, 25:15–25, 1993.

[42] T.-H. Ma and J. Spinrad. Cycle-free partial orders and chordal comparability graphs. *Order*, 8:49–61, 1991.

[43] T. A. McKee and F. R. McMorris. *Topics in Intersection Graph Theory*. SIAM Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia, 1999.

[44] Y. Metelsky and R. Tyshkevich. Line graphs of Helly hypergraphs. *SIAM Journal on Discrete Mathematics*, 16(3):438–448, 2003.

[45] J. H. Muller. *Local structure in graph classes*. PhD thesis, Georgia Institute of Technology, March 1988.

[46] D. Peleg. Proximity-preserving labeling schemes and their applications. In *Graph Theoretic Concepts in Computer Science, Proceedings of the 25th International Workshop (Ascona, Switzerland)*, volume 1665 of *Lecture Notes in Computer Science*, pages 30–41. Springer-Verlag, 1999.

[47] D. Peleg. Informative labeling schemes for graphs. In *Mathematical Foundations of Computer Science, Proceedings of the 25th International-Symposium (Bratislava, Slovakia)*, volume 1893 of *Lecture Notes in Computer Science*, pages 579–588. Springer-Verlag, 2000.

[48] E. Prisner. Bicliques in graphs I: Bounds on their number. *Combinatorica*, 20(1):109–117, 2000.

[49] F. S. Roberts. Indifference graphs. In F. Harary, editor, *Proof Techniques in Graph Theory*, pages 139–146. Academic Press, New York, 1969.

[50] N. D. Roussopoulos. A $\max\{m,n\}$ algorithm for determining the graph H from its line graph G. *Information Processing Letters*, 2:108–112, 1973.

[51] N. Santoro and R. Khatib. Labelling and implicit routing in networks. *The Computer Journal*, 28:5–8, 1985.

[52] J. Spinrad. personal communication.

[53] J. Spinrad. *Efficient Graph Representation*. Fields Institute Monographs. AMS, Providence, 2003.

[54] V. Stix. Finding all maximal cliques in dynamic graphs. *Computational Optimization and Applications*, 27:173–186, 2004.

[55] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. In *42nd Annual Symposium on Foundations of Computer Science (Las Vegas, USA)*, pages 242–251. IEEE, 2001.

[56] M. Thorup and U. Zwick. Compact routing schemes. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures (Heraklion, Greece)*, pages 1–10. ACM, 2001.

[57] D. B. West. *Introduction to Graph Theory, second edition.* Prentice Hall, Toronto, 2000.

[58] H. Whitney. Congruent graphs and the connectivity of graphs. *American Journal of Mathematics*, 54:150–168, 1932.

Appendix A

Definitions

The following are the definitions of a variety of terms pertaining to graph classes seen in this thesis; by no means is this list of definitions meant to be self-contained, rather, the definitions are intended to jog the memory of the reader. Unless otherwise indicated, the definitions are taken from Brandstädt, Le, and Spinrad [8].

H-free A graph is H-free if it does not contain H as an induced subgraph.

Almost tree(k) A graph is an almost tree(k) if there are at most k edges not in a spanning tree of each biconnected component.

Arboricity [31] The arboricity of a graph is the maximum value of $\frac{|E_H|}{|V_H| - 1}$ taken over all vertex induced subgraphs H.

Asteroidal triple A set of three vertices such that, for every pair of the three vertices, there is a path connecting the pair that avoids the neighbourhood of the remaining vertex.

Astral triple A set of three vertices such that, for every pair of the three vertices, there is a path connecting the pair that does not contain two consecutive vertices in the neighbourhood of the remaining vertex.

Autograph [45] A graph G is an autograph if there is a bijection f from V_G to some set S of n positive integers such that $uv \in E_G \iff |f(u) - f(v)| \in S$.

Bandwidth The bandwidth of a graph G is the minimum value for which G is a subgraph of the k^{th} power of $P_{|V_G|}$, the path on $|V_G|$ vertices.

r-bic (defined in Chapter 5) A graph with no vertex in more than r maximal bicliques.

Biclique A complete bipartite subgraph.

Binary tree [57] A rooted tree in which no vertex has more than two children.

Boxicity The boxicity of a graph is the minimum value d for which it is the intersection graph of boxes in d-dimensional space.

Chain graph For simplicity, we define a graph to be a chain graph if it is a P_5-free connected bipartite graph [53]; a more involved definition can be found in [8]

Chordal bipartite graph A bipartite graph which contains no induced cycles of length greater than four.

Chordal graph A graph is chordal if it contains no induced cycles of length greater than three.

Circle graph A graph is a circle graph if it is the intersection graph of some family of chords in a circle.

Circular arc graph A graph is a circular arc graph if it is the intersection graph of some family of arcs of a circle.

Claw A claw is a $K_{1,3}$

Cliquewidth [7] Let $[k]$ denote the set $\{1,\ldots,k\}$ and let $l(v)$ denote the label of a vertex v. A cliquewidth-k graph is defined recursively as follows.

- Any graph G with $V_G = \{v\}$ and $l(v) \in [k]$ is a cliquewidth-k graph.
- Let G_1 and G_2 be cliquewidth-k graphs, and let i and j belong to $[k]$. The following are also cliquewidth-k graphs.
  - The disjoint union of G_1 and G_2.
  - The graph formed from G_1 by switching the labels of all vertices with label i to label j.
  - The graph formed from G_1 by adding all edges v_1v_2, where $l(v_1) = i$ and $l(v_2) = j$.

The cliquewidth of a graph is the minimum value of k for which it is a cliquewidth-k graph.

Cobipartite graph A graph is cobipartite if its complement is bipartite.

Cograph For simplicity, we define a graph to be a cograph if it can be reduced to an edgeless graph by repeatedly taking complements within components [43]; a more involved definition can be found in [8].

Comparability graph A graph is a comparability graph if the edges have a transitive orientation.

Containment class A class $\mathcal{G}$ of graphs is an containment class if there is a set $\mathcal{S}$ of sets such that every graph G in $\mathcal{G}$ is the containment graph of a family $\mathcal{S}'$ of sets from $\mathcal{S}$ (a family allows the sets to occur with repetition).

Containment graph For a given family of sets $\mathcal{S}$, the containment graph of this family is the graph with vertex set $\mathcal{S}$ such that two vertices are adjacent if and only if the set corresponding to one vertex is a subset of the other.

Convex bipartite [45] A bipartite graph with bipartition (X, Y) is convex if, without loss of generality, there is a total order on the vertices of X such that if y in Y is adjacent to x_1 and x_2 in X, then it is also adjacent to all the vertices between x_1 and x_2 in the total ordering.

***k*-decomposable** [31] A graph is k-decomposable if, for all subgraphs H with more than k vertices, there exists k vertices whose deletion causes H to be disconnected with no component containing more than $\frac{2|H|}{3}$ vertices.

Disk intersection graph [53] A graph is a disk intersection graph if it is the intersection graph of some family of disks in the plane.

Distance hereditary A graph is distance hereditary if it is connected and all the induced paths have the same length.

***k*-dot product graph** A graph G is a k-dot product graph if each vertex v can be assigned a vector $\overline{v}$ of length k such that $v_1v_2 \in E_G \iff \overline{v_1} \cdot \overline{v_2} \geq 1$, where $\cdot$ is the standard inner product of two vectors.

EPT graph A graph is an EPT graph if it is the intersection graph of nontrivial simple paths in a tree, where the intersection of paths is considered using edges.

Forest A graph with no cycles.

Genus (of a graph) The genus of a graph is the smallest genus of a surface in which the graph has a crossing-free embedding.

Hereditary property A graph property P is hereditary if, for any graph G satisfying P, every induced subgraph of G satisfies P.

Hereditary degree-*k* graph A graph is hereditary degree-k if each vertex induced subgraph has a vertex of degree at most k.

Hypercube [9] The k-dimensional hypercube is the graph on 2^k vertices, each labelled with a distinct binary string of length k, where two vertices are adjacent if and only if their corresponding strings differ in exactly one position.

Hypergraph A hypergraph H is a pair of sets $(V, \mathcal{E})$, where $\mathcal{E}$ is a family of subsets of V. The rank of H is the value $\max_{e \in \mathcal{E}}\{|e|\}$.

Intersection class A class $\mathcal{G}$ of graphs is an intersection class if there is a set $\mathcal{S}$ of sets such that every graph G in $\mathcal{G}$ is the intersection graph of a family $\mathcal{S}'$ of sets from $\mathcal{S}$ (a family allows the sets to occur with repetition).

Intersection graph For a given family of sets $\mathcal{S}$, the intersection graph of this family is the graph with vertex set $\mathcal{S}$ such that two vertices are adjacent if and only if the intersection of their corresponding sets is nonempty.

Interval graph A graph is an interval graph if it is the intersection graph of some family of intervals on the real line.

k-interval graph A graph is a k-interval graph if it is the intersection graph of some family of sets of k intervals on the real line.

Interval number The interval number of a graph is the smallest number k for which it is a k-interval graph.

Line graph (of a hypergraph) Given a hypergraph $H = (V, \mathcal{E})$, its line graph is the graph $L(H) = (\mathcal{E}, E_{L(H)})$ for which $ee' \in E_{L(H)}$ if and only if $e \neq e'$ and $e \cap e' \neq \emptyset$.

Line graph (of a simple graph) Given a graph $G = (V_G, E_G)$, its line graph is the graph $L(G) = (E_G, E_{L(G)})$ for which $\{u, v\} \in E_{L(G)}$ if and only if u and v are adjacent edges in G.

Mesh As intended by Peleg [47], a mesh is the Cartesian product of two paths.

r-mino (defined in Chapter 5) A graph with no vertex in more than r maximal cliques.

Outdegree-k A graph is an outdegree-k graph if the edges can be oriented such that no vertex has outdegree greater than k.

Outerplanar A graph is outerplanar if it has a crossing-free embedding in the plane such that all vertices are on the same face.

k-outerplanar A graph is 1-outerplanar if it is outerplanar. For $k > 1$, a graph is k-outerplanar provided it has a planar embedding such that if all the vertices on the exterior face are deleted, the connected components of the remaining graph are all $(k-1)$-outerplanar.

Partial order A binary relation is a partial order on a set if it reflexive, transitive, and antisymmetric.

Permutation graph A graph is a permutation graph if it is the intersection graph of some family of lines that intersect two parallel lines.

Planar A graph is planar if it has a crossing-free embedding in the plane.

Poset A poset P is a pair $(V, \preceq)$ for which $\preceq$ is a partial order on V. A poset is often represented by an acyclic digraph.

An ordering $(v_1, \dots, v_n)$ of V is a linear extension of P if, for all $i, j \in \{1, \dots, n\}$, $v_i \preceq v_j \Rightarrow i \leq j$. A family of posets $\cup_{l=0}^{k} P_l$, where $P_l = (V, \preceq_l)$, realizes P if $v_a \preceq v_b \Rightarrow v_a \preceq_l v_b$, for all $i \in \{1, \dots, k\}$. The dimension of P is the smallest number of linear extensions of P that realize P.

Proper interval graph A graph is a proper interval graph if it is the intersection graph of some family of intervals that do not contain one another.

Recursive $r(n)$-separator [24] A class of graphs $\mathcal{G}$ has a recursive $r(n)$-separator if, for every G in $\mathcal{G}$, there exists a subset S of vertices such that $|S| \leq r(|V_G|)$, and every connected component G' of $G \setminus S$ belongs to $\mathcal{G}$ and has at most $\frac{2|V_G|}{3}$ vertices.

Rooted tree A tree which has a single vertex denoted as root. Typically, a rooted tree is considered as a directed graph, where edges are directed away from the root.

Series-parallel A multigraph G is series-parallel if it has an orientation for which, for every pair of edges, G does not contain a cycle that meets the edges in the same direction and another that meets the edges in opposite directions.

k-sparse A graph G is k-sparse if $|E_G| \leq k|V_G|$.

Split A graph is split if there is a partition of its vertices into a clique and an independent set.

Threshold graph A graph is a threshold graph if it is a threshold tolerance graph with a constant tolerance function.

Threshold tolerance graph A graph G is a threshold tolerance graph if there is a weight function $w : V_G \longrightarrow \mathbb{R}^+$ and a tolerance function $t : V_G \longrightarrow \mathbb{R}^+$ such that $uv \in E_G \Longleftrightarrow w_u + w_v \geq \min(t_u, t_v)$.

Torus As intended by Peleg [47], a torus is the Cartesian product of two cycles.

Total graph Given a graph G, its total graph $T(G)$ is defined by $V_{T(G)} = V_G \cup E_G$ and $u, v \in E_G$ if and only u and v are adjacent in G or u and v are incident in G.

Transitive closure of a rooted tree Given a rooted tree T, its transitive closure is the graph G defined by $V_G = V_T$ and $uv \in E_G$ if and only if there is a path from u to v in T, that does not pass through the root.

Treewidth The treewidth of a graph G is the minimum value of $\omega(G') - 1$ taken over all triangulations G' of G, where $\omega(H)$ denotes the size of the largest clique.

Triangle A cycle on three vertices.

Uniformly k-sparse [53] A graph G is said to be uniformly k-sparse if no subgraph H has more than $\frac{k|V_H|(\log|V_G|)}{\log|V_H|}$ edges.

Unit interval graph See proper interval graph.

Vertex induced universal graph A graph G is a vertex induced universal graph of a set of graphs S if all members of S are vertex induced subgraphs of G.

Well (α, g)-separated Given the complexity of this definition, the reader is advised to consult Katz, Katz, and Peleg [35].

Appendix B

Computation Models

The majority of algorithms found in this thesis (markers, decoders, and relabellers) employ a word-level RAM (random access machine; for more on this topic see, for example, Aho, Hopcroft, and Ullman [2]) computational model. Presented below is a summary of three well known computation models, namely, unit-cost RAM, log-cost RAM, and word-level RAM [2, 16, 29], followed by a justification of why the latter was chosen as the computational model for this thesis.

Unit-cost RAM Words can contain an unlimited number of bits, as such, any size input can be represented by a single word. Each operation (addition, multiplication, comparison, memory addressing, bitwise and, bitwise or, etcetera) costs one unit of time.

Log-cost RAM Words can contain an unlimited number of bits, as such, any size input can be represented by a single word. Unlike unit-cost RAM, in which each operation costs one unit of time, the cost of each operation is proportional to the number of bits in the operands; for example, the number n requires $\log n$ bits to store, so it requires $\Theta(\log n)$ time to calculate n^2.

Word-level RAM Algorithms that receive λ bit inputs use $O(\lambda)$ bit words; for example, an algorithm with $O(\log n)$ bit inputs uses $O(\log n)$ bit words. The cost of each operation is proportional to the number of words used by the operands.

The unit-cost RAM model is simple to understand and results in straightforward calculations of the running time of algorithms; however, this model misrepresents the actual time required to perform certain operations, such as multiplication, on large operands. In contrast, the log-cost RAM model accurately represents the performance of a machine on large input; however, this model leads to cumbersome calculations of the running time, as basic operations like memory addressing/pointer referencing cannot be performed in constant time. Both the unit-cost RAM and log-cost RAM models make memory too potent, as an entire data structure can fit in one word.

Word-level RAM offers a tradeoff between unit-cost RAM and log-cost RAM in that the calculation of running times of algorithms remains straightforward, while the size of words does not get unreasonably large. Current publications on data structures often use a word-level RAM computational model; in particular, the majority of papers on informative labelling schemes calculate running times using this model, even if no mention of computation models appears in the paper. Articles on informative labelling schemes that explicitly discuss the use of word-level RAM computation models include Abiteboul, Kaplan, and Milo [1], Alstrup, Gavoille, Kaplan, and Rauhe [5], Alstrup and Rauhe [6], Gavoille and Paul [20], Gavoille and Peleg [23], Gavoille, Katz, Katz, Paul, and Peleg [19], and Kaplan and Milo [32].

Appendix C

Pseudocode

The following is pseudocode that can be used to implement many of the high level algorithms presented in this thesis. In presenting the pseudocode, we try to maintain the convention that stacks are represented using overline notation; for example, $\overline{S}$ would be a stack, whereas S would be a set.

C.1 Line graphs

C.1.1 Deleting a vertex

Recall the algorithm DeleteVertex, found in Figure 4.5, which is used to relabel a line graph when a vertex is deleted. The following pseudocode can be used to implement DeleteVertex.

DeleteVertex($L(G), v$)

Input: An adjacency labelling of a line graph $L(G)$ (that is, the labels thereof) created using our dynamic scheme, and a vertex v in $V_{L(G)}$.

Output: An adjacency labeling of a line graph $L(G')$ (again, the labels thereof) formed by deleting v from $L(G)$.

```
1   for $i \leftarrow 0$ to 1 do
2       if $v.nn_i = 1$ then
3           FreeBase($v.ep_i$)
4       else $\overline{\mathcal{L}_i} \leftarrow$ GetIncidentNeighbors($v, i$)
5           DecrementNN($\overline{\mathcal{L}_i}$)
6           RemoveFromList($v, i$)
7   FreeLine($v$)
```

1: For each i in $\{0, 1\}$, we must determine the effect that the deletion of v has on the endpoint $v.ep_i$ in the base.

2,3: If $v.ep_i$ is incident only with v, then it will become an isolated vertex once v is deleted. The function FreeBase frees the identifier of $v.ep_i$ for future use.

4-6: If $v.ep_i$ is incident with edges other than v, then we must remove v from the circular doubly linked list about $v.ep_i$ and update the label of each edge in this circular doubly linked list.

7: Once all the vertex labels have been changed to reflect the new graph, we delete v using FreeLine, which frees its prelabel for future use.

……

GetIncidentNeighbours($t, tend$)

Input: A pair $(t, tend)$, where t is and edge in the base and $tend$ is a value, either 0 or 1, used to denote and endpoint of t.

Output: A stack $\overline{S}$ consisting of all pairs of the form $(s, send)$, where $s.ep_{send} = t.ep_{tend}$.

```
$\overline{S} \leftarrow$ NIL
$s \leftarrow t$
$send \leftarrow tend$
Push($\overline{S}$, $(s, send)$)
while $s.nx_{send} \neq t$ do
    $s \leftarrow s.nx_{send}$
    $send \leftarrow$ End($s, t.ep_{tend}$)
    Push($\overline{S}$, $(s, send)$)
return $\overline{S}$
```

……

DecrementNN($\overline{S}$)

Input: A stack $\overline{S}$ of pairs of the form $(s, send)$ where s is an edge in the base and $send$ is a value, either 0 or 1, used to denote an endpoint of s.

Output: For each pair $(s, send)$ in $\overline{S}$, DecrementNN decrements the value of $s.nn_{send}$ by one.

```
while $\overline{S} \neq$ Nil do
    $(s, send) \leftarrow$ Pop($\overline{S}$)
    $s.nn_{send} \leftarrow s.nn_{send} - 1$
```

……

RemoveFromList($y, yend$)

Input: A pair $(y, yend)$, where y is an edge in the base graph, and $yend$ is a value, either 0 or 1, which denotes an endpoint of y.

Output: RemoveFromList removes y from the circular doubly linked list about $y.ep_{yend}$.

```
$w \leftarrow y.prev_{yend}$
$z \leftarrow y.nx_{yend}$
$wend \leftarrow$ End($w, y.ep_{yend}$)
$zend \leftarrow$ End($z, y.ep_{yend}$)
$w.nx_{wend} \leftarrow z$
$z.prev_{zend} \leftarrow w$
```

……

End(t, w)

Input: A pair (t, w), where t is an edge of the base that has w as one of its endpoints.

Output: End returns the value of i for which $t.ep_i = w$.

```
if $t.ep_0 = w$ then
    return 0
else return 1
```

C.1.2 Adding a vertex

Recall the algorithm AddVertex, found in Figure 4.6, which is used to relabel a line graph when a vertex is added. The following pseudocode can be used to implement AddVertex.

AddVertex($L(G), X$)

Input: An adjacency labelling of a line graph $L(G)$ created using our dynamic scheme, and a subset X of $V_{L(G)}$.

Output: Let $L(G')$ be the graph formed by adding a new vertex v to $L(G)$, where v is adjacent to exactly those vertices in X. Providing $L(G')$ is a line graph, the output is an adjacency labelling of $L(G')$. If $L(G')$ is not a line graph, the output indicates as such.

```
 1  $v \leftarrow$ GetIdentifierLine()
 2  switch
 3   case $|X| = 0$:
 4      NoNeighbours($v, 0$)
 5      NoNeighbours($v, 1$)
 6   case $|X| \geq 1$:
 7      ($valid_0, vpt_0, valid_1, vpt_1, good$) $\leftarrow$ FindValid($X$)
 8      if $good = 1$ then
 9          EstablishEdgeInBase($valid_0, vpt_0, valid_1, vpt_1, v$)
10      else error this is no longer a line graph
```

1: As the new vertex does not yet have an identifier, the function GetIdentifierLine is used to assign one. Recall that in Section 3.1.3 we assumed that such an identifier could be obtained in O(1) time.

3-5: The new vertex is isolated, so a new isolated edge must be added to the core.

6-10: The new vertex has at least one neighbour, so we must try to find a valid set. If a valid set is found, then we use the valid set to represent the new vertex. Otherwise, if no valid set is found, the new graph is not a line graph.

……

NoNeighbours($t, tend$)

Input: A pair ($t, tend$), where t is an edge in the base and $tend$ is a value, either 0 or 1, used to denote and endpoint of t.

Output: NoNeighbours establishes t as the only edge of the base that is incident with $t.ep_{tend}$.

```
$t.ep_{tend} \leftarrow$ GetIdentifierBase()
$t.nn_{tend} \leftarrow 1$
$t.nx_{tend} \leftarrow t$
$t.prev_{tend} \leftarrow t$
```

……

FindValid(X)

Input: A set X of edges in the base.

Output: The five-tuple ($edge_0, end_0, edge_1, end_1, val$) with values as follows.

- If X has a valid set then val will have value 1, otherwise it will have value 0.
- If X has a valid set of size two, then $edge_0.ep_{end_0}$ and $edge_1.ep_{end_1}$ are the vertices in the valid set. If the valid set is of size one, then the valid set consists of the vertex $edge_0.ep_{end_0}$, where $edge_1 = end_1 =$ NIL.

```
 1  $edge_0 \leftarrow$ some member of $X$
 2  $end_0 \leftarrow 0$
 3  $trychanged_0 \leftarrow 0$
 4  while $end_0 \leq 1$ do
 5      $(X_0, fail) \leftarrow$ ELIMINATE$(X, edge_0, end_0)$
 6      if $fail = 0$ then
 7          if $X_0 = \emptyset$ then
 8              return $(edge_0, end_0,$ NIL, NIL, $1)$
 9          else $edge_1 \leftarrow$ some member of $X_0$
10              $end_1 \leftarrow 0$
11              $trychanged_1 \leftarrow 0$
12              while $end_1 \leq 1$ do
13                  $(X_1, fail) \leftarrow$ ELIMINATE$(X_0, edge_1, end_1)$
14                  if $fail = 0$ and $X_1 = \emptyset$ then
15                      return $(edge_0, end_0, edge_1, end_1, 1)$
16                  else $end_1 \leftarrow end_1 + 1$
17                      if $trychanged_1 = 0$ and $end_1 = 2$ then
18                          $(C, changed) \leftarrow$ CHANGEBASE$(edge_1)$
19                          $trychanged_1 \leftarrow 1$
20                          if $changed = 1$ then
21                              if $edge_0 \in C$ then
22                                  $(C, changed) \leftarrow$ CHANGEBASE$(edge_1)$
23                              else $end_1 \leftarrow 0$
24      $end_0 \leftarrow end_0 + 1$
25      if $end_0 = 2$ then
26          if $trychanged_0 = 1$ then
27              return (NIL, NIL, NIL, NIL, 0)
28          else $(C, changed) \leftarrow$ CHANGEBASE$(edge_0)$
29              $trychanged_0 \leftarrow 1$
30              if $changed = 1$ then
31                  $end_0 \leftarrow 0$
```

1: If X has a valid set then every member of X will have exactly one endpoint in the valid set. As such, we choose a member of X, namely $edge_0$, so as to include one of its endpoints in the valid set.

2: We first try to include $edge_0.ep_0$, in the valid set. If we later determine that $edge_0.ep_0$ cannot be included in any valid set, then we will try to include $edge_0.ep_1$ instead. The value of end_0 indicates whether we are considering $edge_0.ep_0$ or $edge_0.ep_1$.

3: In Chapter 4 we discussed how a component of a line graph can have two bases which are partition not-isomorphic; in particular, for a given set of vertices, one of the bases may yield a valid set while the other may not. It may be necessary to change the base of a component in order to find a valid set. The variable $trychanged_0$ is used to indicate if we have attempted to change the base of the component containing $edge_0$. If $trychanged_0 = 1$, then we have previously attempted to change the base; otherwise, $trychanged_0 = 0$ and we have not tried to change the base. Recall that, when we say that a base is changed, we ultimately mean that the labelling of the line graph has been changed so as to reflect the new base.

4-31: As previously mentioned, we continue to look for a valid set, providing there is at least one of $edge_0.ep_0$ and $edge_0.ep_1$ which we have not tried to include in a valid set.

5: Letting X_0 be the subset of edges in X that are not incident with $edge_0.ep_{end_0}$, we observe that if there is another vertex in the valid set, then it must come from an edge in X_0. In order to determine X_0, FINDVALID uses the function ELIMINATE. If the circular doubly liked list about $edge_0.ep_{end_0}$ contains an edge which is not in X then ELIMINATE will set $fail$ to 1; otherwise, it will set $fail$ to 0 and return X_0.

6-23: If $fail = 0$, then the circular doubly liked list about $edge_0.ep_{end_0}$ did not contain any edges not in X. As such, we may continue trying to place $edge_0.ep_{end_0}$ in the valid set.

7,8: Given that we have not yet found any reason to exclude $edge_0.ep_{end_0}$ from the valid set, if $X_0 = \emptyset$ then all of the edges in X are incident with $edge_0.ep_{end_0}$. Therefore, $\{edge_0.ep_{end_0}\}$ is a valid set.

9-23: If $X_0 \neq \emptyset$, then there are members of X which are not incident with $edge_0.ep_{end_0}$ so we must include a second vertex in the valid set.

9-11: As we did with $edge_0$, we choose an edge $edge_1$ from X_0 and try to include one of its endpoints in the valid set. Like $edge_0$, $edge_1$ has corresponding variables end_1 and $trychanged_1$.

12-23: As we did with $edge_0$, we continue to look for a valid set, providing there is at least one of $edge_1.ep_0$ and $edge_1.ep_1$ which we have not tried to include in the valid set. We first check if $\{edge_0.ep_{end_0}, edge_1.ep_0\}$ is a valid set, if it is not then we will try $\{edge_0.ep_{end_0}, edge_1.ep_1\}$.

13: Letting X_1 be the subset of edges in X_0 that are not incident with $edge_1.ep_{end_1}$, we observe that $\{edge_0.ep_{end_0}, edge_1.ep_{end_1}\}$ is a valid set if and only if $X_1 = \emptyset$ and the circular doubly linked list at $edge_1.ep_{end_1}$ does not contain any edges which are not in X_0. To determine X_1, FINDVALID uses ELIMINATE just as it did to determine X_0.

14-23: If $fail = 0$ and $X_1 = \emptyset$ then $\{edge_0.ep_{end_0}, edge_1.ep_{end_1}\}$ is a valid set. Otherwise, $\{edge_0.ep_{end_0}, edge_1.ep_{end_1}\}$ is not a valid set, so we will need to try another endpoint of $edge_1$ or perhaps another base for the component containing $edge_1$.

17-23: If $end_1 = 2$, then we have already tried to include $edge_1.ep_1$ in the valid set so we must now try changing the base of the component containing $edge_1$. This is only allowed if the base has not been changed, that is, if $trychanged_1 = 0$.

18: To change the base of the component containing $edge_1$, FINDVALID relies on a function called CHANGEBASE. We let C be the set of vertices in the same component as $edge_1$, where *changed* is a variable used to represent whether or not the base was changed.

19: We set $trychanged_1$ to 1 in order to indicate that an attempt was made to change the base of the component containing $edge_1$. The reader should note the distinction between *changed* and *trychanged*; *trychanged* merely indicates that an attempt was made to change the base, whereas *changed* indicates if a change was actually made.

20-23: If the base of the component containing $edge_1$ was changed, there are two possibilities; either C contains $edge_0$ or it does not. If it does, then we change the base of the component containing $edge_1$ back to its original state as we do not wish to change the base of the component containing $edge_0$ at this time; if it does not, then we set end_1 to 0 and repeat the process of trying to include an endpoint of $edge_1$ in the valid set along with $edge_0.ep_{end_0}$.

24-31: If line 24 is reached, then either the call of ELIMINATE in line 6 found an edge in the circular doubly linked list at $edge_0.ep_{end_0}$ which was not in X or, in choosing an edge $edge_1$ from X_0, neither endpoint of $edge_1$ could be put in a valid set with $edge_0.ep_{end_0}$ regardless of the base used to represent the component containing $edge_1$. Either way, $edge_0.ep_{end_0}$ cannot belong to a valid set using the present base. This segment of the algorithm is similar to that involving $edge_1$ in lines 17 though 25.

……

ELIMINATE($T, t, tend$)

Input: A triple $(T, t, tend)$, where T is a set of edges in the base, t is a member of T, and *tend* is a value, either 0 or 1, used to denote an endpoint of t.

Output: Let $\mathcal{L}$ denote the set of edges in the circular linked list about $t.ep_{tend}$. ELIMINATE outputs a pair (T', val), where, if $\mathcal{L} \nsubseteq T$, then val has value 1. Otherwise, if $\mathcal{L} \subseteq T$, then val has value 0, and $T' = T \setminus \mathcal{L}$.

```
$w \leftarrow t$
$wend \leftarrow tend$
$T \leftarrow T \setminus \{w\}$
while $w.nx_{wend} \neq t$ do
    $w \leftarrow w.nx_{wend}$
    $wend \leftarrow$ END$(w, t.ep_{tend})$
    if $w \in T$ then
        $T \leftarrow T \setminus \{w\}$
    else return $(T, 1)$
return $(T, 0)$
```

……

CHANGEBASE(a)

Input: An edge a of the base graph.

Output: Let C be the component of the base containing a. If C does not have another partition non-isomorphic base, CHANGEBASE outputs the pair (*comp*, *changed*), where *changed* has value 0. Otherwise, if C does have another partition non-isomorphic base, CHANGEBASE changes the base of C, and outputs the pair (*comp*, *changed*), where *comp* is the set of vertices in C and *changed* has value 0.

```
$changed \leftarrow 0$
switch
  case $a.nn_0 = 2$ and $a.nn_1 = 2$:
    $b \leftarrow a.nx_0$
    $eb \leftarrow 1 - \textsc{End}(b, a.ep_0)$
    $c \leftarrow a.nx_1$
    $ec \leftarrow 1 - \textsc{End}(c, a.ep_1)$
    if $b.ep_{eb} = c.ep_{ec}$ then
      if $b.nn_{eb} = 2$ then
        $\textsc{SwitchK3ToK13}(a, b, c)$
        $comp \leftarrow \{a, b, c\}$
        $changed \leftarrow 1$
      elseif $b.nn_{eb} = 3$ then
        if $c = b.nx_{eb}$ then
          $d \leftarrow c.nx_{ec}$
        else $d \leftarrow b.nx_{eb}$
        $ed \leftarrow 1 - \textsc{End}(d, b.ep_{eb})$
        if $d.nn_{ed} = 1$ then
          $\textsc{Switch}(a, d)$
          $comp \leftarrow \{a, b, c, d\}$
          $changed \leftarrow 1$
  case ($a.nn_0 = 3$ and $a.nn_1 = 1$) or ($a.nn_0 = 1$ and $a.nn_1 = 3$):
    if $a.nn_0 = 3$ then
      $ea = 0$
    else $ea = 1$
    $b \leftarrow a.nx_{ea}$
    $eb \leftarrow 1 - \textsc{End}(b, a.ep_{ea})$
    $c \leftarrow b.nx_{1-eb}$
    $ec \leftarrow 1 - \textsc{End}(c, a.ep_{ea})$
    if $b.nn_{eb} = 1$ and $c.nn_{ec} = 1$ then
      $\textsc{SwitchK13ToK3}(a, b, c)$
      $comp \leftarrow \{a, b, c\}$
      $changed \leftarrow 1$
    elseif $b.nn_{eb} = 2$ and $c.nn_{ec} = 2$ then
      $d \leftarrow b.nx_{eb}$
      $ed \leftarrow 1 - \textsc{End}(d, b.ep_{eb})$
      if $d.ep_{ed} = c.ep_{ec}$ then
        $\textsc{Switch}(a, d)$
        $comp \leftarrow \{a, b, c, d\}$
        $changed \leftarrow 1$
  case ($a.nn_0 = 3$ and $a.nn_1 = 2$) or ($a.nn_0 = 2$ and $a.nn_1 = 3$):
    if $a.nn_0 = 3$ then
      $ea = 0$
    else $ea = 1$
    $b \leftarrow a.nx_{ea}$
    $eb \leftarrow 1 - \textsc{End}(b, a.ep_{ea})$
    $c \leftarrow b.nx_{1-eb}$
    $ec \leftarrow 1 - \textsc{End}(c, a.ep_{ea})$
    $f \leftarrow a.nx_{1-ea}$
    $ef \leftarrow 1 - \textsc{End}(f, a.ep_{1-ea})$
    if $f.ep_{ef} = b.ep_{eb}$ then
      if $b.nn_{eb} = 2$ and $c.nn_{ec} = 1$ then
        $\textsc{Switch}(c, f)$
        $comp \leftarrow \{a, b, c, f\}$
        $changed \leftarrow 1$
      elseif $b.nn_{eb} = 3$ and $c.nn_{ec} = 2$ then
        $d = c.nx_{ec}$
        $ed \leftarrow \textsc{End}(d, c.ep_{ec})$
        if $d.ep_{1-ed} = b.ep_{eb}$ then
          $\textsc{Switch}(c, f)$
          $comp \leftarrow \{a, b, c, d, f\}$
          $changed \leftarrow 1$
    elseif $f.ep_{ef} = c.ep_{ec}$ then
      if $b.nn_{eb} = 1$ and $c.nn_{ec} = 2$ then
```

```
65                  SWITCH$(b, f)$
66                  $comp \leftarrow \{a, b, c, f\}$
67                  $changed \leftarrow 1$
68              elseif $b.nn_{eb} = 2$ and $c.nn_{ec} = 3$ then
69                  $d = b.nx_{eb}$
70                  $ed \leftarrow 1 -$ END$(d, b.ep_{eb})$
71                  if $d.ep_{ed} = c.ep_{ec}$ then
72                      SWITCH$(b, f)$
73                      $comp \leftarrow \{a, b, c, d, f\}$
74                      $changed \leftarrow 1$
75      case $a.nn_0 = 3$ and $a.nn_1 = 3$:
76          $b \leftarrow a.nx_0$
77          $eb \leftarrow 1 -$ END$(b, a.ep_0)$
78          $c \leftarrow b.nx_{1-eb}$
79          $ec \leftarrow 1 -$ END$(c, a.ep_0)$
80          $f \leftarrow a.nx_1$
81          $ef \leftarrow 1 -$ END$(f, a.ep_1)$
82          $h \leftarrow f.nx_{1-ef}$
83          $eh \leftarrow 1 -$ END$(h, a.ep_1)$
84          if $f.ep_{ef} = b.ep_{eb}$ and $h.ep_{eh} = c.ep_{ec}$ then
85              if $b.nn_{eb} = 2$ and $c.nn_{ec} = 2$ then
86                  SWITCH$(c, f)$
87                  $comp \leftarrow \{a, b, c, f, h\}$
88                  $changed \leftarrow 1$
89              elseif $b.nn_{eb} = 3$ and $c.nn_{ec} = 3$ then
90                  if $f = b.nx_{eb}$ then
91                      $d \leftarrow f.nx_{ef}$
92                  else $d \leftarrow b.nx_{eb}$
93                  $ed \leftarrow 1 -$ END$(d, b.ep_{eb})$
94                  if $d.ep_{ed} = c.ep_{ec}$ then
95                      SWITCH$(c, f)$
96                      $comp \leftarrow \{a, b, c, d, f, h\}$
97                      $changed \leftarrow 1$
98          elseif $f.ep_{ef} = c.ep_{ec}$ and $h.ep_{eh} = b.ep_{eb}$ then
99              if $b.nn_{eb} = 2$ and $c.nn_{ec} = 2$ then
100                 SWITCH$(c, h)$
101                 $comp \leftarrow \{a, b, c, f, h\}$
102                 $changed \leftarrow 1$
103             elseif $b.nn_{eb} = 3$ and $c.nn_{ec} = 3$ then
104                 if $h = b.nx_{eb}$ then
105                     $d \leftarrow h.nx_{eh}$
106                 else $d \leftarrow b.nx_{eb}$
107                 $ed \leftarrow 1 -$ END$(d, b.ep_{eb})$
108                 if $d.ep_{ed} = c.ep_{ec}$ then
109                     SWITCH$(c, h)$
110                     $comp \leftarrow \{a, b, c, d, f, h\}$
111                     $changed \leftarrow 1$
112     return $(comp, changed)$
```

1: As mentioned, the variable *changed* is used to indicate if the base of C has been changed. The default value of *changed* is 0 and will be set to 1 when the base of C is changed.

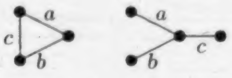
2-112: In determining if the base of C can be changed, we consider a series of cases based upon the degrees of the endpoints of a.

3-21: In this case, each endpoint of a is incident with exactly one other edge besides a.

4-7: We let b and c be the edges, other than a, incident with $a.ep_0$ and $a.ep_1$, respectively. The endpoints $b.ep_{eb}$ and $c.ep_{ec}$ are set to be the endpoints of b and c, respectively, which are furthest from a.

8-21: Either $b.ep_{eb}$ and $c.ep_{ec}$ are the same vertex or they are not. If they are not, then the condition $a.nn_0 = a.nn_1 = 2$ guarantees that the base of C has an induced P_4 or C_4. Since none of the bases of the graphs found in Theorem 4.2 (that is, the graphs in Figure 4.1(b)) has an induced P_4 or C_4, the base of C cannot be changed.

9-12: If $b.nn_{eb} = 2$, then $b.ep_{eb}$ is incident only with b and c. Thereby, the base of C is the K_3 shown in Figure C.1(a) so we use the function SWITCHK3TOK13 to change it to the $K_{1,3}$ shown in Figure C.1(b).



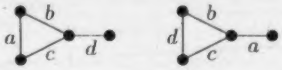
(a) (b)

Figure C.1: Two partition non-isomorphic bases of C

13-21: If $b.nn_{eb} = 3$, then $b.ep_{eb}$ is incident with b, c, and another vertex which we will call d. Observe that if $b.nn_{eb} > 3$, then the base of C cannot be changed as none of the graphs found in in Figure 4.1(b) has a vertex of degree greater than three.

15-17: We ensure that d is distinct from b and c, then set $d.ep_{ed}$ to be the endpoint of d that is furthest from b.

18-21: If $d.nn_{ed} = 1$, then the base of C is as shown in Figure C.2(a). Using the function SWITCH, we change the base of C to the graph depicted in Figure C.2(b). Furthermore, observe that if $d.nn_{ed} > 1$, then the conditions $a.nn_0 = 2$ and $b.nn_{eb} = 3$ guarantee that the base of C has an induced P_4 which prevents the base from being changed.



(a) (b)

Figure C.2: Two partition non-isomorphic bases of C

22-40: We now consider the case when one endpoint of a is incident with two additional edges besides a, and the other endpoint is incident with only a itself.

23-25: We set $a.ep_{ea}$ to be the endpoint of a with degree three.

26-29: We let b and c be the edges, other than a, that are incident with $a.ep_{ea}$. Moreover, we let $b.ep_{eb}$ and $c.ep_{ec}$ be the endpoints of b and c, respectively, that are furthest from a.

30-40: Given that $a.nn_{1\text{-}ea} = 1$ and $a.nn_{ea} = 3$, the only way that the base of C can be changed is if $b.nn_{eb} = c.nn_{ec} = 1$ or if $b.nn_{eb} = c.nn_{ec} = 2$. Otherwise, the the base of C has an induced P_4 which prevents it from being changed.

30-33: In this case, the base of C is the $K_{1,3}$ shown in Figure C.1(b), so we change it to the K_3 shown in Figure C.1(a).

34-40: In this case, both $b.ep_{eb}$ and $c.ep_{ec}$ are incident with another edge besides b and c, respectively.

35,36: We let d be the edge, other than b, that is incident with $b.ep_{eb}$. Moreover, we let $d.ep_{ed}$ be the endpoint of d that is furthest from b.

37-40: If $d.ep_{ed} \neq c.ep_{ec}$, then the condition $a.nn_{1\text{-}ea} = 1$ guarantees that the base of C has an induced P_4 which prevents it from being changed. On the other hand, if $d.ep_{ed} = c.ep_{ec}$, then the base of C is as shown in Figure C.2(b) so we change it to the base shown in Figure C.2(a).

41-74: We now consider the case when one endpoint of a is incident with two additional edges besides a, and the other endpoint is incident with one additional edge besides a.

42-44: We set $a.ep_{ea}$ to be the endpoint of a with degree three.

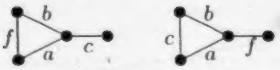
45-48: We let b and c be the edges, other than a, that are incident with $a.ep_{ea}$. Moreover, we let $b.ep_{eb}$ and $c.ep_{ec}$ be the endpoints of b and c, respectively, that are furthest from a.

49,50: We let f be the edge, other than a, that is incident with $a.ep_{1\text{-}ea}$. Moreover, we let $f.ep_{ef}$ be the endpoint of f that is furthest from a.

51-74: If neither $b.ep_{eb} = f.ep_{ef}$, nor $c.ep_{ec} = f.ep_{ef}$, then the base of C has an induced P_4 which prevents it from being changed.

52-62: Given that $a.nn_{ea} = 3$, the only way that the base of C can be changed is if $b.nn_{eb} = 2$ and $c.nn_{ec} = 1$ or if $b.nn_{eb} = 3$ and $c.nn_{ec} = 2$. Otherwise, the the base of C has an induced P_4 which prevents it from being changed.

52-55: If $b.nn_{eb} = 2$ and $c.nn_{ec} = 1$, then there are no additional edges in the graph. The base of C is as shown in Figure C.3(a) so we change it to the base shown in Figure C.3(b).



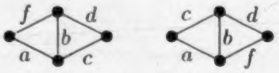
(a) (b)

Figure C.3: Two partition non-isomorphic bases of C

56-62: In this case, both $b.ep_{eb}$ and $c.ep_{ec}$ are incident with another edge besides b and c, respectively.

57-58: We let d be the edge, other than c, that is incident with $c.ep_{ec}$. Moreover, we let $d.ep_{ed}$ be the endpoint of d that is closest to c.

59-62: If $d.ep_{1\text{-}ed} \neq b.ep_{eb}$, then the condition $a.nn_{ea} = 3$ guarantees that the base of C has an induced P_4 which prevents it from being changed. On the other hand, if $d.ep_{1\text{-}ed} = b.ep_{eb}$, then the base of C is as shown in Figure C.4(a), so we change it to the base shown in Figure C.4(b).



(a) (b)

Figure C.4: Two partition non-isomorphic bases of C

63-74: This case is analogous to that found in lines 51 through 62, except that $c.ep_{ec} = f.ep_{ef}$, not $b.ep_{eb} = f.ep_{ef}$.

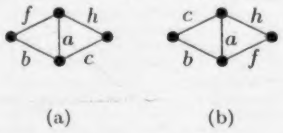
75-111: We now consider the case when both endpoints of a are incident with two additional edges besides a.

76-83: We let b and c be the edges, other than a, that are incident with $a.ep_0$. Moreover, we let $b.ep_{eb}$ and $c.ep_{ec}$ be the endpoints of b and c, respectively, that are furthest from a. The edges f and h are defined similarly for $a.ep_1$.

84-111: If $b.ep_{eb} \neq f.ep_{ef}$ or $c.ep_{ec} \neq h.ep_{eh}$, and $b.ep_{eb} \neq h.ep_{eh}$ or $c.ep_{ec} \neq f.ep_{ef}$ then the base of C has an induced P_4 or C_4 which prevents it from being changed.

84-97: Given that $a.nn_0 = a.nn_1 = 3$, the only way that the base of C can be changed is if $b.nn_{eb} = c.nn_{ec} = 2$ or if $b.nn_{eb} = c.nn_{ec} = 3$. Otherwise, the base of C has an induced P_4 which prevents it from being changed.

85-88: If $b.nn_{eb} = c.nn_{ec} = 2$, then there are no additional edges in the graph. The base of C is as shown in Figure C.5(a), so we change it to the base shown in Figure C.5(b).



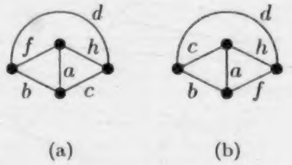
(a) (b)

Figure C.5: Two partition non-isomorphic bases of C

89-97: In this case, both $b.ep_{eb}$ and $c.ep_{ec}$ are incident with another edge besides b and c, respectively.

91-93: We let d be the edge, other than b and f, that is incident with $b.ep_{eb}$. Moreover, we let $d.ep_{ed}$ be the endpoint of d that furthest from b.

94-97: If $d.ep_{1\text{-}ed} \neq c.ep_{ec}$, then the base of C has an induced P_4 which prevents it from being changed. On the other hand, if $d.ep_{1\text{-}ed} = c.ep_{ec}$, then the base of C is as shown in Figure C.6(a), so we change it to the base shown in Figure C.6(b).



(a) (b)

Figure C.6: Two partition non-isomorphic bases of C

98-111: This case is analogous to that found in lines 84 through 97, except that $c.ep_{ec} = f.ep_{ef}$ and $b.ep_{eb} = h.ep_{eh}$ not $b.ep_{eb} = f.ep_{ef}$ and $c.ep_{ec} = h.ep_{eh}$.

158: When ChangeBase is finished it returns the pair (*comp*, *changed*).

……

SwitchK3ToK13(w, y, z)

Input: A triple (w, y, z) of edges in the base that constitute a component in the form of a K_3.

Output: SwitchK3ToK13 changes the labels of w, y, and z so that they form a $K_{1,3}$.

```
FreeBase($w.ep_0$)
FreeBase($w.ep_1$)
FreeBase($y.ep_0$)
FreeBase($y.ep_1$)
FreeBase($z.ep_0$)
FreeBase($z.ep_1$)
```

```
$w.ep_0 \leftarrow$ GetIdentifierBase()
$w.ep_1 \leftarrow$ GetIdentifierBase()
$y.ep_0 \leftarrow w.ep_0$
$y.ep_1 \leftarrow$ GetIdentifierBase()
$z.ep_0 \leftarrow w.ep_0$
$z.ep_1 \leftarrow$ GetIdentifierBase()
$w.nx_0 \leftarrow z$
$w.prev_0 \leftarrow y$
$y.nx_0 \leftarrow w$
$y.prev_0 \leftarrow z$
$z.nx_0 \leftarrow y$
$z.prev_0 \leftarrow w$
$w.nx_1 \leftarrow w$
$w.prev_1 \leftarrow w$
$y.nx_1 \leftarrow y$
$y.prev_1 \leftarrow y$
$z.nx_1 \leftarrow z$
$z.prev_1 \leftarrow z$
$w.nn_0 \leftarrow 3$
$y.nn_0 \leftarrow 3$
$z.nn_0 \leftarrow 3$
$w.nn_1 \leftarrow 1$
$y.nn_1 \leftarrow 1$
$z.nn_1 \leftarrow 1$
```

……

SwitchK13ToK3(w, y, z)

Input: A triple (w, y, z) of edges in the base that constitute a component in the form of a $K_{1,3}$.

Output: SwitchK13ToK3 changes the labels of w, y, and z so that they form a K_3.

```
FreeBase($w.ep_0$)
FreeBase($w.ep_1$)
FreeBase($y.ep_0$)
FreeBase($y.ep_1$)
FreeBase($z.ep_0$)
FreeBase($z.ep_1$)
$w.ep_0 \leftarrow$ GetIdentifierBase()
$w.ep_1 \leftarrow$ GetIdentifierBase()
$y.ep_0 \leftarrow w.ep_0$
$y.ep_1 \leftarrow$ GetIdentifierBase()
$z.ep_0 \leftarrow w.ep_1$
$z.ep_1 \leftarrow y.ep_1$
$w.nx_0 \leftarrow y$
$w.prev_0 \leftarrow y$
$y.nx_0 \leftarrow w$
$y.prev_0 \leftarrow w$
$y.nx_1 \leftarrow z$
$y.prev_1 \leftarrow z$
$z.nx_1 \leftarrow y$
$z.prec_1 \leftarrow y$
$w.nx_1 \leftarrow z$
$w.prev_1 \leftarrow z$
$z.nx_0 \leftarrow w$
$z.prev_0 \leftarrow w$
$w.nn_0 \leftarrow 2$
$y.nn_0 \leftarrow 2$
$z.nn_0 \leftarrow 2$
$w.nn_1 \leftarrow 2$
$y.nn_1 \leftarrow 2$
$z.nn_1 \leftarrow 2$
```

……

Switch(w_0, w_1)

Input: A pair (w_0, w_1) of edges in the base whose component has a partition non-isomorphic base that can be formed by switching w_0 and w_1.

Output: SWITCH changes base of the component containing w_0 and w_1, by switching w_0 and w_1.

```
1   for $j \leftarrow 0$ to 1 do
2       for $i \leftarrow 0$ to 1 do
3           $temp_j \leftarrow$ GETIDENTIFIERLINE()
4           INSERTINTOLIST($w_j, i, temp_j, i$)
5           $temp_j.nn_i \leftarrow w_j.nn_i$
6           REMOVEFROMLIST($w_j, i$)
7   for $j \leftarrow 0$ to 1 do
8       for $i \leftarrow 0$ to 1 do
9           INSERTINTOLIST($temp_j, i, w_{1-j}, i$)
10          $w_{1-j}.nn_i \leftarrow temp_j.nn_i$
11          REMOVEFROMLIST($temp_j, i$)
12          FREEBASE($temp_j.ep_i$)
```

1-6: For each j in $\{0, 1\}$, we replace w_j with a temporary edge $temp_j$.

7-12: For each j in $\{0, 1\}$, we replace $temp_j$ with w_{1-j}.

……

ESTABLISHEDGEINBASE($edge_0, end_0, edge_1, end_1, t$)

Input: A 5-tuple $(edge_0, end_0, edge_1, end_1, t)$, where t is a vertex of the line graph and, for i in $\{0, 1\}$, end_i is an endpoint of $edge_i$, an edge of the base. It is permissible for $edge_1$ to have value NIL, in which case end_1 will also have value NIL.

Output: Provided $edge_1 \neq$ NIL, ESTABLISHEDGEINBASE changes the vertex labels to reflect the addition of the edge t between vertices $edge_0.ep_{end_0}$ and $edge_1.ep_{end_1}$ of the base. If $edge_1 =$ NIL, ESTABLISHEDGEINBASE creates a new vertex in the base and changes the vertex labels to reflect the addition of the edge t between vertex $edge_0.ep_{end_0}$ and the new vertex.

```
if $edge_1 =$ NIL then
    $size \leftarrow 1$
    NONEIGHBOURS($t, 1$)
else $size \leftarrow 2$
for $i \leftarrow 0$ to $size - 1$ do
    INSERTINTOLIST($edge_i, end_i, t, i$)
    $\overline{S} \leftarrow$ GETINCIDENTNEIGHBORS($t, i$)
    INCREMENTNN($\overline{S}$)
```

……

INSERTINTOLIST($w, wend, y, yend$)

Input: A 4-tuple $(w, wend, y, yend)$, where w and y are edges of the base graph, and $wend$ and $yend$ are values, either 0 or 1, which denote endpoints of w and y, respectively.

Output: INSERTINTOLIST adds y to the circular doubly linked list of edges about $w.ep_{wend}$, such that $y.ep_{yend} = w.ep_{wend}$.

```
$y.ep_{yend} \leftarrow w.ep_{wend}$
$z \leftarrow w.nx_{wend}$
$zend \leftarrow$ END($z, w.ep_{wend}$)
$w.nx_{wend} \leftarrow y$
$y.prev_{yend} \leftarrow w$
$y.nx_{yend} \leftarrow z$
$z.prev_{zend} \leftarrow y$
```

……

INCREMENTNN($\overline{S}$)

Input: A stack $\overline{S}$ of pairs of the form $(s, send)$ where s is an edge in the base and $send$ is a value, either 0 or 1, used to denote an endpoint of s.

Output: For each pair $(s, send)$ in $\overline{S}$, INCREMENTNN increments the value of $s.nn_{send}$ by one.

```
while $\overline{S} \neq$ NIL do
    $(s, send) \leftarrow$ POP($\overline{S}$)
    $s.nn_{send} \leftarrow s.nn_{send} + 1$
```

C.1.3 Deleting an edge

Recall the algorithm DELETEEDGE, found in Figure 4.7, which is used to relabel a line graph when an edge is deleted. The following pseudocode can be used to implement DELETEEDGE.

DELETEEDGE(a, b)

Input: An adjacency labelling of a line graph $L(G)$ created using our dynamic scheme, and two distinct vertices a and b of $V_{L(G)}$ for which $ab \in E_{L(G)}$.

Output: An adjacency labeling of a graph $L(G')$ formed by deleting the edge ab from $L(G)$, providing $L(G')$ is a line graph. If $L(G')$ is not a line graph, then the output indicates as such.

```
for $k \leftarrow 0$ to 1 do
    for $l \leftarrow 0$ to 1 do
        if $a.ep_k = b.ep_l$ then
            $ea \leftarrow k$
            $eb \leftarrow l$
switch
 case $a.nn_{ea} = 2$:
    CASEAC()
 case $a.nn_{ea} = 3$:
    if $a.nx_{ea} = b$ then
        $c \leftarrow b.nx_{eb}$
    else $c \leftarrow a.nx_{ea}$
    $ec \leftarrow 1 -$ END($c, a.ep_{ea}$)
    switch
     case $c.nn_{ec} = 1$
        CASEBD()
     case $c.nn_{ec} = 2$
        $f \leftarrow c.nx_{ec}$
        $ef \leftarrow 1 -$ END($f, c.ep_{ec}$)
        switch
         case $f.ep_{ef} = a.ep_{1-ea}$
            switch
             case $f.nn_{ef} \geq 4$
                error this is no longer a line graph
             case $f.nn_{ef} = 3$:
                if $f.nx_{ef} = a$ then
                    $g \leftarrow a.nx_{1-ea}$
                else $g \leftarrow f.nx_{ef}$
                $eg \leftarrow 1 -$ END($g, f.ep_{ef}$)
                if $g.nn_{eg} = 1$ then
                    CASEF()
                else error this is no longer a line graph
             case $f.nn_{ef} = 2$:
```

```
          CaseE()
      case $f.ep_{ef} = b.ep_{1\text{-}eb}$
        switch
          case $f.nn_{ef} \geq 4$
            error this is no longer a line graph
          case $f.nn_{ef} = 3$:
            if $f.nx_{ef} = b$ then
              $g \leftarrow b.nx_{1\text{-}eb}$
            else $g \leftarrow f.nx_{ef}$
            $eg \leftarrow 1 - \textsc{End}(g, f.ep_{ef})$
            if $g.nn_{eg} = 1$ then
              CaseFSymmetric()
            else error this is no longer a line graph
          case $f.nn_{ef} = 2$:
            CaseESymmetric()
      case $f.ep_{ef} = a.ep_{1\text{-}ea}$ and $f.ep_{ef} = b.ep_{1\text{-}eb}$:
        error this is no longer a line graph
    case $c.nn_{ec} = 3$:
      if $a.nn_{1\text{-}ea} \neq 2$ or $b.nn_{1\text{-}eb} \neq 2$ then
        error this is no longer a line graph
      else $f \leftarrow a.nx_{1\text{-}ea}$
        $h \leftarrow b.nx_{1\text{-}eb}$
      $ef \leftarrow \textsc{End}(f, a.ep_{1\text{-}ea})$
      $eh \leftarrow \textsc{End}(h, b.ep_{1\text{-}eb})$
      if $f.ep_{1\text{-}ef} \neq c.ep_{ec}$ or $h.ep_{1\text{-}eh} \neq c.ep_{ec}$ then
        error this is no longer a line graph
      else CaseG()
    case $c.nn_{ec} > 3$:
      error this is no longer a line graph
case $a.nn_{ea} = 4$:
  if $a.nx_{ea} \neq b$ then
    $c \leftarrow a.nx_{ea}$
  else $c \leftarrow b.nx_{eb}$
  $ec \leftarrow 1 - \textsc{End}(c, a.ep_{ea})$
  if $a.nx_{ec} \neq b$ or $a.nx_{ec} \neq c$ then
    $i \leftarrow a.nx_{ea}$
  elseif $b.nx_{eb} \neq a$ or $b.nx_{eb} \neq c$ then
    $i \leftarrow b.nx_{eb}$
  else $i \leftarrow c.nx_{1\text{-}ec}$
  $ei \leftarrow 1 - \textsc{End}(i, a.ep_{ea})$
  switch
    case $c.nn_{ec} > 2$ or $i.nn_{ei} > 2$
      error this is no longer a line graph
    case $c.nn_{ec} = 1$ and $i.nn_{ei} = 1$
      if $a.nn_{1\text{-}ea} = 1$ then
        CaseH()
      elseif $b.nn_{1\text{-}eb} = 1$ then
        CaseHSymmetric()
      else error this is no longer a line graph
    case $c.nn_{ec} = 2$ and $i.nn_{ei} = 2$
      $f \leftarrow c.nx_{ec}$
      $g \leftarrow i.nx_{ei}$
      $ef \leftarrow 1 - \textsc{End}(f, c.ep_{ec})$
      $eg \leftarrow \textsc{End}(g, i.ep_{ei})$
      if $f.ep_{ef} = g.ep_{1\text{-}eg} = a.ep_{1\text{-}ea}$ and $f.nn_{ef} = 3$ then
        CaseJ()
      elseif $f.ep_{ef} = g.ep_{1\text{-}eg} = b.ep_{1\text{-}eb}$ and $f.nn_{ef} = 3$ then
        CaseJSymmetric()
      else error this is no longer a line graph
    case $i.nn_{ei} = 1$ and $c.nn_{ec} = 2$:
      $f \leftarrow c.nx_{ec}$
      $ef \leftarrow 1 - \textsc{End}(c, c.ep_{ec})$
      if $f.ep_{ef} = a.ep_{1\text{-}ea}$ and $f.nn_{ef} = 2$ then
        CaseI()
```

```
98          elseif $f.ep_{ef} = b.ep_{1\text{-}eb}$ and $f.nn_{ef} = 2$ then
99              CaseISymmetric()
100         else error this is no longer a line graph
101       case $i.nn_{ei} = 2$ and $c.nn_{ec} = 1$:
102         $g \leftarrow i.nx_{ei}$
103         $eg \leftarrow$ End($g, i.ep_{ei}$)
104         if $g.ep_{1\text{-}eg} = b.ep_{1\text{-}eb}$ and $g.nn_{1\text{-}eg} = 2$ then
105             CaseISymmetric()
106         elseif $g.ep_{1\text{-}eg} = a.ep_{1\text{-}ea}$ and $g.nn_{1\text{-}eg} = 2$ then
107             CaseISymmetric()
108         else error this is no longer a line graph
109   case $a.nn_{ea} > 4$
110     error this is no longer a line graph
```

1-5: We must first determine the vertex of the base at which a and b intersect. In particular, we set $a.ep_{ea}$ and $b.ep_{eb}$ to be the endpoints at which a and b intersect. Later we will introduce the variables ec, ef, eg, eh, ei, ej, to denote particular endpoints of c, f, g, h, i, and j, respectively.

6-110: We determine the structure of the base surrounding a and b through a series of case statements that allow us to determine which of the cases in Table 4.1 we must deal with.

7,8: If the only edges of the base incident with $a.ep_{ea}$ are a and b themselves, then we are dealing with case A or C. Both cases are handled by the function CaseAC.

9-62: There is exactly one additional edge incident with $a.ep_{ea}$ other than a and b themselves. We call this edge c and set $c.ep_{ec}$ to be the endpoint of c closest to a.

10-12: We determine c using the circular doubly linked list at $a.ep_{ea}$.

15,16: If c is the only edge incident with $c.ep_{ec}$, then we are dealing with case B or D. Both cases are handled by the function CaseBD.

17-50: There is exactly one other edge incident with $c.ep_{ec}$, other than c itself. We call this edge f and set $f.ep_{ef}$ to be the endpoint of f closest to c. The possibility exists that the modified graph is not a line graph, however, if it is a line graph, then we are dealing with case E or F, or symmetric variants thereof.

21-34: The edge f is adjacent to both c and a where, in particular, $f.ep_{ef} = a.ep_{1\text{-}ea}$. Again, the possibility exists that the modified graph is not a line graph, however, if it is a line graph, then we are dealing with case E or F.

23,24: If there are more than three edges incident with $f.ep_{ef}$ then the modified graph is not a line graph, as the structure of G does not resemble any of the cases depicted in Table 4.1.

25-32: There is exactly one other edge incident with $f.ep_{ef}$, other than a and f themselves. We call this edge g and set $g.ep_{eg}$ to be the endpoint of g closest to f. The possibility exists that the modified graph is not a line graph, however, if it is a line graph, then we are dealing with case F.

30-32: If g is the only edge incident with $g.ep_{eg}$, then we are dealing with case F, which is handled by the function CaseF. Otherwise, the modified graph is not a line graph.

33-34: If the only edges incident with $f.ep_{ef}$ are a and f themselves, then we are dealing with case E, which is handled by the function CaseE.

35-48: The edge f is adjacent to both c and b where, in particular, $f.ep_{ef} = b.ep_{1\text{-}eb}$. This case is analogous to that found in lines 21 through 34.

49,50: The edge f is adjacent to c, but neither a nor b. Consequently, the modified graph is not a line graph.

51-60: There are exactly two additional edges incident with $c.ep_{ec}$ other than c itself. If the modified graph is still a line graph, then the base must resemble case G.

52,53: The base resembles case G only if $a.nn_{1\text{-}ea} = b.nn_{1\text{-}eb} = 2$. Otherwise, the modified graph is not a line graph.

54-57: Providing $a.nn_{1\text{-}ea} = b.nn_{1\text{-}eb} = 2$, we let f be the edge incident with $a.ep_{1\text{-}ea}$, other than a, and we let h be the edge incident with $b.ep_{1\text{-}eb}$, other than b. Moreover, we set $f.ep_{ef}$ to be the endpoint of f furthest from a, and set $h.ep_{eh}$ to be the endpoint of f furthest from b.

58-60: The base resembles case G if and only if $f.ep_{1\text{-}ef} = h.ep_{1\text{-}eh} = c.ep_{ec}$. Providing this condition holds, it is handled by the function CaseG; it it does not hold, then the modified graph is not a line graph.

61,62: If there are more than three edges incident with $c.ep_{ec}$ then the modified graph is not a line graph.

63-108: There are exactly two additional edges incident with $a.ep_{ea}$, besides a and b themselves. The possibility exists that the modified graph is not a line graph, however, if it is then we are dealing with cases H, I, or J, or symmetric variants thereof.

64-73: We let c and i be the edges incident with $a.ep_{ea}$, other than a and b themselves. The circular linked list at $a.ep_{ea}$ is used to determine c and i where, moreover, we set $c.ep_{ec}$ and $i.ep_{ei}$ to be the endpoints of c and i, respectively, that are closest to a.

74-76: We require that $c.ep_{ec}$ and $i.ep_{ei}$ be incident with at most one edge other than c and i themselves, otherwise, the modified graph is not a line graph.

77-82: We first consider when $c.ep_{ec}$ and $i.ep_{ei}$ are incident only with c and i, respectively. The possibility exists that the modified graph is not a line graph, however, if it is then we are dealing with case H or a symmetric variant thereof.

78-82: The modified graph is a line graph if and only if at least one of $a.nn_{1\text{-}ea} = 1$ or $b.nn_{1\text{-}eb} = 1$. If $a.nn_{1\text{-}ea} = 1$, then we are dealing with case H, which is handled by the function CASEH. If $a.nn_{1\text{-}ea} > 1$, but $b.nn_{1\text{-}eb} = 1$, then the situation is symmetric to case H.

83-92: We now consider when $c.ep_{ec}$ and $i.ep_{ei}$ are both incident with exactly one edge in addition to c and i, respectively. The possibility exists that the modified graph is not a line graph, however, if it is a line graph, then we are dealing with case J or a symmetric variant thereof.

84-87: We let f and g be the edges incident with $c.ep_{ec}$ and $i.ep_{ei}$, respectively, other than c and i themselves. The circular linked lists at $c.ep_{ec}$ and $i.ep_{ei}$ are used to determine f and g where, moreover, we set $f.ep_{ef}$ to be the endpoint of f furthest from c, and set $g.ep_{eg}$ to be the endpoint of g closest to i.

88-92: The modified graph is a line graph if and only if either $f.ep_{ef} = g.ep_{1\text{-}eg} = b.ep_{1\text{-}eb}$, where $f.nn_{ef} = 3$, or $f.ep_{ef} = g.ep_{1\text{-}eg} = a.ep_{1\text{-}ea}$, where $f.nn_{ef} = 3$. If the latter holds then we are dealing with Case J, which is handled by the function CASEJ. If the latter does not hold but the former does, then the situation is symmetric to Case J.

93-100: We now consider when $i.ep_{ei}$ is incident only with i itself and $c.ep_{ec}$ is incident with exactly one edge other than c. The possibility exists that the modified graph is not a line graph, however, if it is a line graph, then we are dealing with case I or a symmetric variant thereof.

94,95: We let f be the edge incident with $c.ep_{ec}$, other than c itself. The circular linked list at $c.ep_{ec}$ is used to determine f where $f.ep_{ef}$ is set to be the endpoint of f closest to c.

96-100: The modified graph is a line graph if and only if either $f.ep_{ef} = a.ep_{1\text{-}ea}$, where $a.nn_{1\text{-}ea} = 2$, or $f.ep_{ef} = b.ep_{1\text{-}eb}$, where $b.nn_{1\text{-}eb} = 2$. If the latter holds, then we are dealing with Case I, which is handled by the function CASEI. If the latter does not hold, but the former does, then the situation is symmetric to Case I.

101-108: We now consider when $c.ep_{ec}$ is incident only with c itself and $i.ep_{ei}$ is incident with exactly one edge other than i. This case is similar to that found in lines 93 through 100.

109,110: Finally, we consider when there are exactly at least three additional edges incident with $a.ep_{ea}$, other than a and b themselves. In this case, the modified graph is not a line graph.

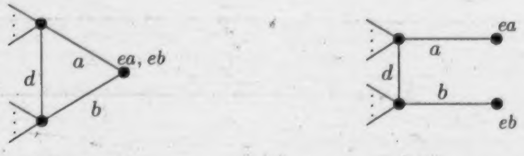
……

CaseAC()

Input: None, however, the algorithm will work globally on the labels seen in DeleteEdge.

Output: CaseAC relabels the vertices of the component containing a and b to reflect the transitions illustrated in Figures C.7 and C.8.

```
FreeBase($a.ep_{ea}$)
FreeBase($b.ep_{eb}$)
$a.ep_{ea} \leftarrow$ GetIdentifierBase()
$b.ep_{eb} \leftarrow$ GetIdentifierBase()
$a.nx_{ea} \leftarrow a$
$a.prev_{ea} \leftarrow a$
$b.nx_{eb} \leftarrow b$
$b.prev_{eb} \leftarrow b$
$a.nn_{ea} \leftarrow 1$
$a.nn_{1\text{-}ea} \leftarrow a.nn_{1\text{-}ea}$
$b.nn_{eb} \leftarrow 1$
$b.nn_{1\text{-}eb} \leftarrow b.nn_{1\text{-}eb}$
```



(a) G (b) G'

Figure C.7: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case A of Table 4.1). The vertices labelled ex in G are as prescribed in the algorithm DeleteEdge; the vertices labelled ex in G' are as prescribed in the algorithm CaseAC

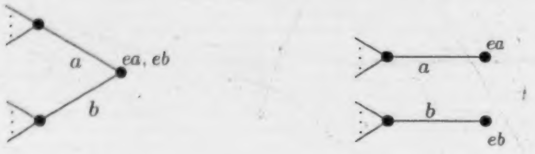
……

CaseBD()

Input: None, however, the algorithm will work globally on the labels seen in DeleteEdge.

Output: CaseBD relabels the vertices of the component containing a and b to reflect the transitions illustrated in Figures C.9 and C.10.

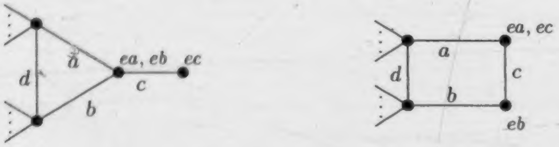
```
FreeBase($a.ep_{ea}$)
FreeBase($b.ep_{eb}$)
FreeBase($c.ep_{ec}$)
FreeBase($c.ep_{1\text{-}ec}$)
$a.ep_{ea} \leftarrow$ GetIdentifierBase()
```



(a) G (b) G'

Figure C.8: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case C of Table 4.1). The vertices labelled ex in G are as prescribed in the algorithm DELETEEDGE; the vertices labelled ex in G' are as prescribed in the algorithm CASEAC

```
$b.ep_{eb} \leftarrow$ GETIDENTIFIERBASE()
$c.ep_{ec} \leftarrow a.ep_{ea}$
$c.ep_{1\text{-}ec} \leftarrow b.ep_{eb}$
$a.nx_{ea} \leftarrow c$
$a.prev_{ea} \leftarrow c$
$c.nx_{ec} \leftarrow a$
$c.prev_{ec} \leftarrow a$
$b.nx_{eb} \leftarrow c$
$b.prev_{eb} \leftarrow c$
$c.nx_{1\text{-}ec} \leftarrow b$
$c.prev_{1\text{-}ec} \leftarrow b$
$a.nn_{ea} \leftarrow 2$
$a.nn_{1\text{-}ea} \leftarrow a.nn_{1\text{-}ea}$
$b.nn_{eb} \leftarrow 2$
$b.nn_{1\text{-}eb} \leftarrow b.nn_{1\text{-}eb}$
$c.nn_{ec} \leftarrow 2$
$c.nn_{1\text{-}ec} \leftarrow 2$
```



(a) G (b) G'

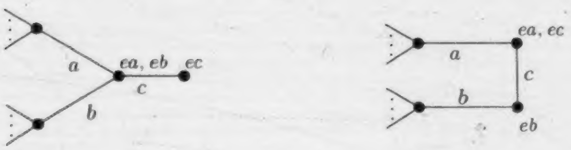
Figure C.9: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case B of Table 4.1). The vertices labelled ex in G are as prescribed in the algorithm DELETEEDGE; the vertices labelled ex in G' are as prescribed in the algorithm CASEBD

……

CASEE()

Input: None, however, the algorithm will work globally on the labels seen in DELETEEDGE.

Output: CASEE relabels the vertices of the component containing a and b to reflect the transition illustrated in Figure C.11.



(a) G (b) G'

Figure C.10: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case D of Table 4.1). The vertices labelled ex in G are as prescribed in the algorithm DELETEEDGE; the vertices labelled ex in G' are as prescribed in the algorithm CASEBD

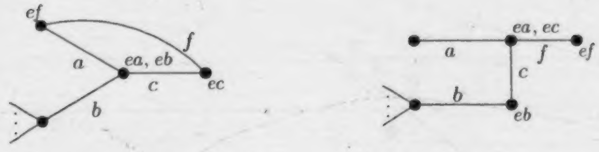
```
FREEBASE($a.ep_{ea}$)
FREEBASE($a.ep_{1\text{-}ea}$)
FREEBASE($b.ep_{eb}$)
FREEBASE($c.ep_{ec}$)
FREEBASE($c.ep_{1\text{-}ec}$)
FREEBASE($f.ep_{ef}$)
FREEBASE($f.ep_{1\text{-}ef}$)
$a.ep_{ea} \leftarrow$ GETIDENTIFIERBASE()
$a.ep_{1\text{-}ea} \leftarrow$ GETIDENTIFIERBASE()
$b.ep_{eb} \leftarrow$ GETIDENTIFIERBASE()
$f.ep_{ef} \leftarrow$ GETIDENTIFIERBASE()
$c.ep_{ec} \leftarrow a.ep_{ea}$
$c.ep_{1\text{-}ec} \leftarrow b.ep_{eb}$
$f.ep_{1\text{-}ef} \leftarrow a.ep_{ea}$
$b.nx_{eb} \leftarrow c$
$b.prev_{eb} \leftarrow c$
$c.nx_{1\text{-}ec} \leftarrow b$
$c.prev_{1\text{-}ec} \leftarrow b$
$c.nx_{ec} \leftarrow a$
$c.prev_{ec} \leftarrow f$
$a.nx_{ea} \leftarrow f$
$a.prev_{ea} \leftarrow c$
$f.nx_{1\text{-}ef} \leftarrow c$
$f.prev_{1\text{-}ef} \leftarrow a$
$f.nx_{ef} \leftarrow f$
$f.prev_{ef} \leftarrow f$
$a.nx_{1\text{-}ea} \leftarrow a$
$a.prev_{1\text{-}ea} \leftarrow a$
$a.nn_{ea} \leftarrow 3$
$a.nn_{1\text{-}ea} \leftarrow 1$
$b.nn_{eb} \leftarrow 2$
$b.nn_{1\text{-}eb} \leftarrow b.nn_{1\text{-}eb}$
$c.nn_{ec} \leftarrow 3$
$c.nn_{1\text{-}ec} \leftarrow 2$
$f.nn_{ef} \leftarrow 1$
$f.nn_{1\text{-}ef} \leftarrow 3$
```

……

CASEESYMMETRIC()

Input: None, however, the algorithm will work globally on the labels seen in DELETEEDGE.

Output: CASEESYMMETRIC relabels the vertices of the component containing a and b to reflect the transition illustrated in Figure C.12.



(a) G (b) G'

Figure C.11: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case E of Table 4.1). The vertices labelled ex in G are as prescribed in the algorithm DeleteEdge; the vertices labelled ex in G' are as prescribed in the algorithm CaseE

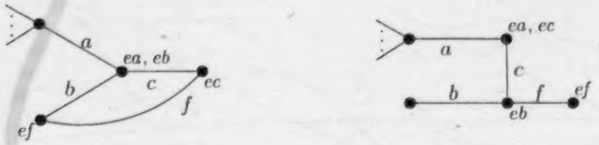
```
FreeBase($a.ep_{ea}$)
FreeBase($b.ep_{eb}$)
FreeBase($b.ep_{1\text{-}eb}$)
FreeBase($c.ep_{ec}$)
FreeBase($c.ep_{1\text{-}ec}$)
FreeBase($f.ep_{ef}$)
FreeBase($f.ep_{1\text{-}ef}$)
$a.ep_{ea} \leftarrow$ GetIdentifierBase()
$b.ep_{eb} \leftarrow$ GetIdentifierBase()
$b.ep_{1\text{-}eb} \leftarrow$ GetIdentifierBase()
$f.ep_{ef} \leftarrow$ GetIdentifierBase()
$c.ep_{ec} \leftarrow a.ep_{ea}$
$c.ep_{1\text{-}ec} \leftarrow b.ep_{eb}$
$f.ep_{1\text{-}ef} \leftarrow a.ep_{ea}$
$b.nx_{eb} \leftarrow c$
$b.prev_{eb} \leftarrow f$
$c.nx_{1\text{-}ec} \leftarrow f$
$c.prev_{1\text{-}ec} \leftarrow b$
$f.nx_{1\text{-}ef} \leftarrow b$
$f.prev_{1\text{-}ef} \leftarrow c$
$c.nx_{ec} \leftarrow a$
$c.prev_{ec} \leftarrow a$
$a.nx_{ea} \leftarrow c$
$a.prev_{ea} \leftarrow c$
$f.nx_{ef} \leftarrow f$
$f.prev_{ef} \leftarrow f$
$b.nx_{1\text{-}eb} \leftarrow b$
$b.prev_{1\text{-}eb} \leftarrow b$
$a.nn_{ea} \leftarrow 2$
$a.nn_{1\text{-}ea} \leftarrow a.nn_{1\text{-}ea}$
$b.nn_{eb} \leftarrow 3$
$b.nn_{1\text{-}eb} \leftarrow 1$
$c.nn_{ec} \leftarrow 2$
$c.nn_{1\text{-}ec} \leftarrow 3$
$f.nn_{ef} \leftarrow 1$
$f.nn_{1\text{-}ef} \leftarrow 3$
```

……

CaseF()

Input: None, however, the algorithm will work globally on the labels seen in DeleteEdge.

Output: CaseF relabels the vertices of the component containing a and b to reflect the transition illustrated in Figure C.13.

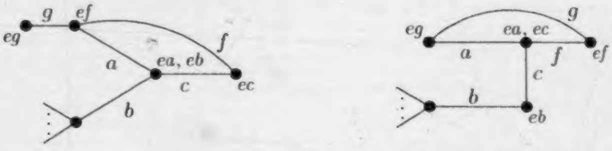


(a) G (b) G'

Figure C.12: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (symmetric to case E of Table 4.1). The vertices labelled ex in G are as prescribed in the algorithm DELETEEDGE; the vertices labelled ex in G' are as prescribed in the algorithm CASEESYMMETRIC

```
FREEBASE($a.ep_{ea}$)
FREEBASE($a.ep_{1\text{-}ea}$)
FREEBASE($b.ep_{eb}$)
FREEBASE($c.ep_{ec}$)
FREEBASE($c.ep_{1\text{-}ec}$)
FREEBASE($f.ep_{ef}$)
FREEBASE($f.ep_{1\text{-}ef}$)
FREEBASE($g.ep_{eg}$)
FREEBASE($g.ep_{1\text{-}eg}$)
$a.ep_{ea} \leftarrow$ GETIDENTIFIERBASE()
$a.ep_{1\text{-}ea} \leftarrow$ GETIDENTIFIERBASE()
$b.ep_{eb} \leftarrow$ GETIDENTIFIERBASE()
$f.ep_{ef} \leftarrow$ GETIDENTIFIERBASE()
$c.ep_{ec} \leftarrow a.ep_{ea}$
$c.ep_{1\text{-}ec} \leftarrow b.ep_{eb}$
$f.ep_{1\text{-}ef} \leftarrow a.ep_{ea}$
$g.ep_{eg} \leftarrow a.ep_{1\text{-}ea}$
$g.ep_{1\text{-}eg} \leftarrow f.ep_{ef}$
$b.nx_{eb} \leftarrow c$
$b.prev_{eb} \leftarrow c$
$c.nx_{1\text{-}ec} \leftarrow b$
$c.prev_{1\text{-}ec} \leftarrow b$
$c.nx_{ec} \leftarrow a$
$c.prev_{ec} \leftarrow f$
$a.nx_{ea} \leftarrow f$
$a.prev_{ea} \leftarrow c$
$f.nx_{1\text{-}ef} \leftarrow c$
$f.prev_{1\text{-}ef} \leftarrow a$
$f.nx_{ef} \leftarrow g$
$f.prev_{ef} \leftarrow g$
$g.nx_{1\text{-}eg} \leftarrow f$
$g.prev_{1\text{-}eg} \leftarrow f$
$g.nx_{eg} \leftarrow a$
$g.prev_{eg} \leftarrow a$
$a.nx_{1\text{-}ea} \leftarrow g$
$a.prev_{1\text{-}ea} \leftarrow g$
$a.nn_{ea} \leftarrow 3$
$a.nn_{1\text{-}ea} \leftarrow 2$
$b.nn_{eb} \leftarrow 2$
$b.nn_{1\text{-}eb} \leftarrow b.nn_{1\text{-}eb}$
$c.nn_{ec} \leftarrow 3$
$c.nn_{1\text{-}ec} \leftarrow 2$
```

```
$f.nn_{ef} \leftarrow 2$
$f.nn_{1\text{-}ef} \leftarrow 3$
$g.nn_{eg} \leftarrow 2$
$g.nn_{1\text{-}eg} \leftarrow 2$
```



(a) G (b) G'

Figure C.13: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case F of Table 4.1). The vertices labelled ex in G are as prescribed in the algorithm DELETEEDGE; the vertices labelled ex in G' are as prescribed in the algorithm CASEF

……

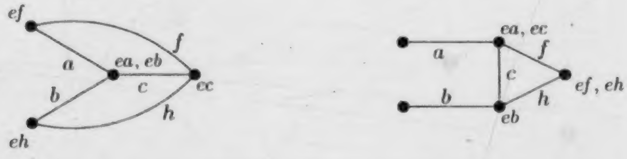
CASEG()

Input: None, however, the algorithm will work globally on the labels seen in DELETEEDGE.

Output: CASEG relabels the vertices of the component containing a and b to reflect the transition illustrated in Figure C.14.

```
FREEBASE($a.ep_{ea}$)
FREEBASE($a.ep_{1\text{-}ea}$)
FREEBASE($b.ep_{eb}$)
FREEBASE($c.ep_{ec}$)
FREEBASE($c.ep_{1\text{-}ec}$)
FREEBASE($f.ep_{ef}$)
FREEBASE($f.ep_{1\text{-}ef}$)
FREEBASE($h.ep_{eh}$)
FREEBASE($h.ep_{1\text{-}eh}$)
$a.ep_{ea} \leftarrow$ GETIDENTIFIERBASE()
$a.ep_{1\text{-}ea} \leftarrow$ GETIDENTIFIERBASE()
$b.ep_{eb} \leftarrow$ GETIDENTIFIERBASE()
$b.ep_{1\text{-}eb} \leftarrow$ GETIDENTIFIERBASE()
$f.ep_{ef} \leftarrow$ GETIDENTIFIERBASE()
$c.ep_{ec} \leftarrow a.ep_{ea}$
$c.ep_{1\text{-}ec} \leftarrow b.ep_{1\text{-}eb}$
$f.ep_{1\text{-}ef} \leftarrow a.ep_{ea}$
$h.ep_{eh} \leftarrow f.ep_{ef}$
$h.ep_{1\text{-}eh} \leftarrow b.ep_{eb}$
$b.nx_{eb} \leftarrow c$
$b.prev_{eb} \leftarrow h$
$c.nx_{1\text{-}ec} \leftarrow h$
$c.prev_{1\text{-}ec} \leftarrow b$
$h.nx_{1\text{-}eh} \leftarrow b$
$h.prev_{1\text{-}eh} \leftarrow c$
$c.nx_{ec} \leftarrow a$
$c.prev_{ec} \leftarrow f$
$a.nx_{ea} \leftarrow f$
$a.prev_{ea} \leftarrow c$
$f.nx_{1\text{-}ef} \leftarrow c$
$f.prev_{1\text{-}ef} \leftarrow a$
```

```
$f.nx_{ef} \leftarrow h$
$f.prev_{ef} \leftarrow h$
$h.nx_{eh} \leftarrow f$
$h.prev_{eh} \leftarrow f$
$a.nx_{1\text{-}ea} \leftarrow a$
$a.prev_{1\text{-}ea} \leftarrow a$
$b.nx_{1\text{-}eb} \leftarrow b$
$b.prev_{1\text{-}eb} \leftarrow b$
$a.nn_{ea} \leftarrow 3$
$a.nn_{1\text{-}ea} \leftarrow 1$
$b.nn_{eb} \leftarrow 3$
$b.nn_{1\text{-}eb} \leftarrow 1$
$c.nn_{ec} \leftarrow 3$
$c.nn_{1\text{-}ec} \leftarrow 3$
$f.nn_{ef} \leftarrow 2$
$f.nn_{1\text{-}ef} \leftarrow 3$
$h.nn_{eh} \leftarrow 2$
$h.nn_{1\text{-}eh} \leftarrow 3$
```



(a) G (b) G'

Figure C.14: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case G of Table 4.1). The vertices labelled ex in G are as prescribed in the algorithm DELETEEDGE; the vertices labelled ex in G' are as prescribed in the algorithm CASEG

……

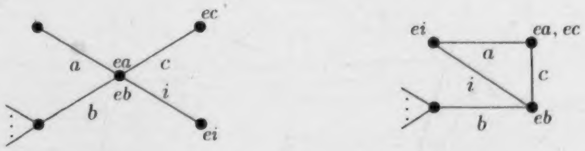
CASEH()

Input: None, however, the algorithm will work globally on the labels seen in DELETEEDGE.

Output: CASEH relabels the vertices of the component containing a and b to reflect the transition illustrated in Figure C.15.

```
FREEBASE($a.ep_{ea}$)
FREEBASE($a.ep_{1\text{-}ea}$)
FREEBASE($b.ep_{eb}$)
FREEBASE($c.ep_{ec}$)
FREEBASE($c.ep_{1\text{-}ec}$)
FREEBASE($i.ep_{ei}$)
FREEBASE($i.ep_{1\text{-}ei}$)
$a.ep_{ea} \leftarrow$ GETIDENTIFIERBASE()
$a.ep_{1\text{-}ea} \leftarrow$ GETIDENTIFIERBASE()
$b.ep_{eb} \leftarrow$ GETIDENTIFIERBASE()
$c.ep_{ec} \leftarrow a.ep_{ea}$
$c.ep_{1\text{-}ec} \leftarrow b.ep_{eb}$
$i.ep_{ei} \leftarrow a.ep_{1\text{-}ea}$
$i.ep_{1\text{-}ei} \leftarrow b.ep_{1\text{-}eb}$
$b.nx_{eb} \leftarrow i$
$b.prev_{eb} \leftarrow c$
$i.nx_{1\text{-}ei} \leftarrow c$
$i.prev_{1\text{-}ei} \leftarrow b$
```

```
$c.nx_{1\text{-}ec} \leftarrow b$
$c.prev_{1\text{-}ec} \leftarrow i$
$c.nx_{ec} \leftarrow a$
$c.prev_{ec} \leftarrow a$
$a.nx_{ea} \leftarrow c$
$a.prev_{ea} \leftarrow c$
$i.nx_{ei} \leftarrow a$
$i.prev_{ei} \leftarrow a$
$a.nx_{1\text{-}ea} \leftarrow i$
$a.prev_{1\text{-}ea} \leftarrow i$
$a.nn_{ea} \leftarrow 2$
$a.nn_{1\text{-}ea} \leftarrow 2$
$b.nn_{eb} \leftarrow 3$
$b.nn_{1\text{-}eb} \leftarrow b.nn_{1\text{-}eb}$
$c.nn_{ec} \leftarrow 2$
$c.nn_{1\text{-}ec} \leftarrow 3$
$i.nn_{ei} \leftarrow 2$
$i.nn_{1\text{-}ei} \leftarrow 3$
```



(a) G (b) G'

Figure C.15: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case H of Table 4.1). The vertices labelled ex in G are as prescribed in the algorithm DELETEEDGE; the vertices labelled ex in G' are as prescribed in the algorithm CASEH

……

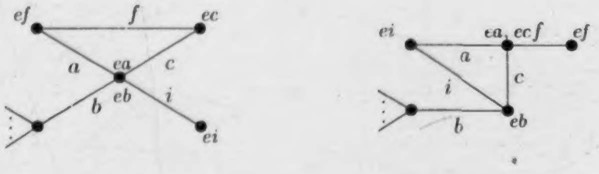
CASEI()

Input: None, however, the algorithm will work globally on the labels seen in DELETEEDGE.

Output: CASEI relabels the vertices of the component containing a and b to reflect the transition illustrated in Figure C.16.

```
FREEBASE($a.ep_{ea}$)
FREEBASE($a.ep_{1\text{-}ea}$)
FREEBASE($b.ep_{eb}$)
FREEBASE($c.ep_{ec}$)
FREEBASE($c.ep_{1\text{-}ec}$)
FREEBASE($f.ep_{ef}$)
FREEBASE($f.ep_{1\text{-}ef}$)
FREEBASE($i.ep_{ei}$)
FREEBASE($i.ep_{1\text{-}ei}$)
$a.ep_{ea} \leftarrow$ GETIDENTIFIERBASE()
$a.ep_{1\text{-}ea} \leftarrow$ GETIDENTIFIERBASE()
$b.ep_{eb} \leftarrow$ GETIDENTIFIERBASE()
$f.ep_{ef} \leftarrow$ GETIDENTIFIERBASE()
$c.ep_{ec} \leftarrow a.ep_{ea}$
$c.ep_{1\text{-}ec} \leftarrow b.ep_{eb}$
$i.ep_{ei} \leftarrow a.ep_{1\text{-}ea}$
$i.ep_{1\text{-}ei} \leftarrow b.ep_{1\text{-}eb}$
$f.ep_{1\text{-}ef} \leftarrow a.ep_{ea}$
```

```
$b.nx_{eb} \leftarrow i$
$b.prev_{eb} \leftarrow c$
$i.nx_{1\text{-}ei} \leftarrow c$
$i.prev_{1\text{-}ei} \leftarrow b$
$c.nx_{1\text{-}ec} \leftarrow b$
$c.prev_{1\text{-}ec} \leftarrow i$
$a.nx_{1\text{-}ea} \leftarrow i$
$a.prev_{1\text{-}ea} \leftarrow i$
$i.nx_{ei} \leftarrow a$
$i.prev_{ei} \leftarrow a$
$c.nx_{ec} \leftarrow a$
$c.prev_{ec} \leftarrow f$
$a.nx_{ea} \leftarrow f$
$a.prev_{ea} \leftarrow c$
$f.nx_{1\text{-}ef} \leftarrow c$
$f.prev_{1\text{-}ef} \leftarrow a$
$f.nx_{ef} \leftarrow f$
$f.prev_{ef} \leftarrow f$
$a.nn_{ea} \leftarrow 3$
$a.nn_{1\text{-}ea} \leftarrow 2$
$b.nn_{eb} \leftarrow 3$
$b.nn_{1\text{-}eb} \leftarrow b.nn_{1\text{-}eb}$
$c.nn_{ec} \leftarrow 3$
$c.nn_{1\text{-}ec} \leftarrow 3$
$f.nn_{ef} \leftarrow 1$
$f.nn_{1\text{-}ef} \leftarrow 3$
$i.nn_{ei} \leftarrow 2$
$i.nn_{1\text{-}ei} \leftarrow 3$
```



(a) G

(b) G'

Figure C.16: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case I of Table 4.1). The vertices labelled ex in G are as prescribed in the algorithm DeleteEdge; the vertices labelled ex in G' are as prescribed in the algorithm CaseI

CaseJ()

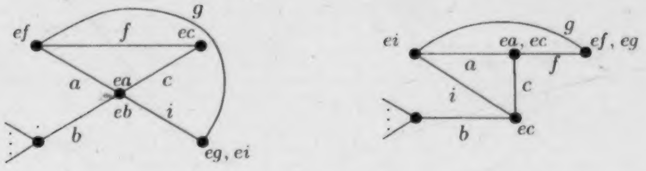
Input: None, however, the algorithm will work globally on the labels seen in DeleteEdge.

Output: CaseJ relabels the vertices of the component containing a and b to reflect the transition illustrated in Figure C.17.

```
FreeBase($a.ep_{ea}$)
FreeBase($a.ep_{1\text{-}ea}$)
FreeBase($b.ep_{eb}$)
FreeBase($c.ep_{ec}$)
FreeBase($c.ep_{1\text{-}ec}$)
FreeBase($f.ep_{ef}$)
FreeBase($f.ep_{1\text{-}ef}$)
FreeBase($g.ep_{eg}$)
FreeBase($g.ep_{1-eg}$)
FreeBase($i.ep_{ei}$)
FreeBase($i.ep_{1-ei}$)
$a.ep_{ea} \leftarrow$ GetIdentifierBase()
$a.ep_{1-ea} \leftarrow$ GetIdentifierBase()
$b.ep_{eb} \leftarrow$ GetIdentifierBase()
$f.ep_{ef} \leftarrow$ GetIdentifierBase()
$c.ep_{ec} \leftarrow a.ep_{ea}$
$c.ep_{1-ec} \leftarrow b.ep_{eb}$
$i.ep_{ei} \leftarrow a.ep_{1-ea}$
$i.ep_{1-ei} \leftarrow b.ep_{1-eb}$
$f.ep_{1-ef} \leftarrow a.ep_{ea}$
$g.ep_{eg} \leftarrow f.ep_{ef}$
$g.ep_{1-eg} \leftarrow i.ep_{ei}$
$b.nx_{eb} \leftarrow i$
$b.prev_{eb} \leftarrow c$
$i.nx_{1-ei} \leftarrow c$
$i.prev_{1-ei} \leftarrow b$
$c.nx_{1-ec} \leftarrow b$
$c.prev_{1-ec} \leftarrow i$
$c.nx_{ec} \leftarrow a$
$c.prev_{ec} \leftarrow f$
$a.nx_{ea} \leftarrow f$
$a.prev_{ea} \leftarrow c$
$f.nx_{1-ef} \leftarrow c$
$f.prev_{1-ef} \leftarrow a$
$i.nx_{ei} \leftarrow g$
$i.prev_{ei} \leftarrow a$
$g.nx_{1-eg} \leftarrow a$
$g.prev_{1-eg} \leftarrow i$
$a.nx_{1-ea} \leftarrow i$
$a.prev_{1-ea} \leftarrow g$
$g.nx_{eg} \leftarrow f$
$g.prev_{eg} \leftarrow f$
$f.nx_{ef} \leftarrow g$
$f.prev_{ef} \leftarrow g$
$a.nn_{ea} \leftarrow 3$
$a.nn_{1-ea} \leftarrow 3$
$b.nn_{eb} \leftarrow 3$
$b.nn_{1-eb} \leftarrow b.nn_{1-eb}$
$c.nn_{ec} \leftarrow 3$
$c.nn_{1-ec} \leftarrow 3$
$f.nn_{ef} \leftarrow 2$
$f.nn_{1-ef} \leftarrow 3$
$g.nn_{eg} \leftarrow 2$
$g.nn_{1-eg} \leftarrow 3$
$i.nn_{ei} \leftarrow 3$
$i.nn_{1-ei} \leftarrow 3$
```

C.1.4 Adding an edge

As discussed in Section 4.2.2, given that the addition of an edge is the opposite of the deletion of an edge, the pseudocode required to add an edge would comprise a case analysis similar to that presented in Section C.1.3.



(a) G (b) G'

Figure C.17: Deleting the edge $\{a, b\}$ from the line graph $L(G)$ (case J of Table 4.1). The vertices labelled ex in G are as prescribed in the algorithm DELETEEDGE; the vertices labelled ex in G' are as prescribed in the algorithm CASEJ

C.2 r-minoes

C.2.1 Deleting a vertex

Recall the algorithm DELETEVERTEX, found in Figure 5.2, which is used to relabel a r-mino when a vertex is deleted. The following pseudocode can be used to implement DELETEVERTEX.

```
DELETEVERTEX($G, v$)
Input: An adjacency labelling of an $r$-mino $G$ created using our dynamic scheme, and a
vertex $v$ in $V_G$. Here we presume that $r \in O(1)$, thereby ensuring error-detection.
Output: An adjacency labeling of an $r$-mino $G'$ formed by deleting $v$ from $G$.

1   if $v.cl_1.nx = v$ then
2       FREECLIQUE($v.cl_1.num$)
3   for $i \leftarrow 1$ to $v.cin$ do
4       $(C, \overline{C}) \leftarrow$ GETCLIQUEMEMBERS($v, i$)
5       $\overline{C'} \leftarrow$ NIL
6       while $\overline{C} \neq$ NIL do
7           $(x, xcl) \leftarrow$ POP($\overline{C}$)
8           if $x \neq v$ then
9               PUSH($\overline{C'}, x$)
10      if GETCOMMONCLIQUES($\overline{C'}$) $= \{v.cl_i.num\}$ then
11          REMOVEFROMCLIQUE($v, i$)
12      else ELIMINATECLIQUE($v, i$)
13  FREEVERTEX($v$)
```

1-2: If v is an isolated vertex, then we free the identifier of the maximal clique $\{v\}$ for future use.

3-9: For each maximal clique C containing v, we obtain the clique $C' = C \setminus \{v\}$.

10-12: If the members of C' share another maximal clique besides C, then we eliminate C. Otherwise, we simply remove v from C.

13: We free the identifier of v for future use.

……

GETCLIQUEMEMBERS(t, tcl)

Input: A pair (t, tcl), where t is a vertex of G and tcl is an index between 1 and $t.cin$.

Output: Let S denote the set of all vertices in the maximal clique $t.cl_{tcl}.num$. GETCLIQUEMEMBERS returns the pair $(S, \overline{S})$, where $\overline{S}$ is a stack consisting of all pairs of the form (s, scl), where $s.cl_{scl}.num = t.cl_{tcl}.num$.

```
$S \leftarrow \emptyset$
$\overline{S} \leftarrow$ NIL
$s \leftarrow t$
$scl \leftarrow tcl$
PUSH($\overline{S}, (s, scl)$)
$S \leftarrow S \cup \{s\}$
while $s.cl_{scl}.nx.id \neq t$ do
    $y \leftarrow s.cl_{scl}.nx.id$
    $scl \leftarrow s.cl_{scl}.nx.index$
    $s \leftarrow y$
    PUSH($\overline{S}, (s, scl)$)
    $S \leftarrow S \cup \{s\}$
return $(S, \overline{S})$
```

……

GETCOMMONCLIQUES($\overline{S}$)

Input: A non-empty stack $\overline{S}$ of vertices.

Output: For each vertex s in $\overline{S}$, let $\mathcal{C}_s$ denote the set of maximal cliques containing s. GETCOMMONCLIQUES returns $\bigcap_{s \in \overline{S}} \mathcal{C}_s$.

```
$s \leftarrow$ POP($\overline{S}$)
$(A, \overline{A}) \leftarrow$ GETCLIQUES($\{s\}$)
while $\overline{S} \neq$ NIL do
    $s \leftarrow$ POP($\overline{S}$)
    $(W, \overline{W}) \leftarrow$ GETCLIQUES($\{s\}$)
    $A \leftarrow A \cap W$
return $A$
```

……

GETCLIQUES(S)

Input: A non-empty set of vertices S.

Output: Let $\mathcal{C}_s$ denote the set of all maximal cliques containing a vertex s, and let $\mathcal{C}$ denote $\bigcup_{s \in S} \mathcal{C}_s$. GETCLIQUES returns the pair $(\mathcal{C}, \overline{T})$, where T is a stack containing an element of the form (t, i), for each entry $t.cl_i.num$ of $\mathcal{C}$.

```
$T \leftarrow \emptyset$
$\overline{T} \leftarrow$ NIL
for $s \in S$ do
    for $i \leftarrow 1$ to $s.cin$ do
        if $s.cl_i.num \notin T$ then
            $T \leftarrow T \cup \{xs.cl_i.num\}$
            PUSH($\overline{T}, (s, i)$)
return $(T, \overline{T})$
```

……

REMOVEFROMCLIQUE(y, ycl)

Input: A pair (y, ycl), where y is a vertex of G and ycl is an index between 1 and $y.cin$. Moreover, $y.cl_{ycl}.nx.id \neq y$.

Output: RemoveFromClique eliminates y from the maximal clique $y.cl_{ycl}.num$.

```
$w \leftarrow y.cl_{ycl}.prev.id$
$wcl \leftarrow y.cl_{ycl}.prev.index$
$z \leftarrow y.cl_{ycl}.nx.id$
$zcl \leftarrow y.cl_{ycl}.nx.index$
$y.cl_{ycl} \leftarrow$ NIL
$w.cl_{xcl}.nx.id \leftarrow z$
$w.cl_{xcl}.nx.index \leftarrow zcl$
$z.cl_{zcl}.prev.id \leftarrow w$
$z.cl_{zcl}.prev.index \leftarrow wcl$
```

……

EliminateClique(t, tcl)

Input: A pair (t, tcl), where t is a vertex of G and tcl is an index between 1 and $t.cin$.

Output: EliminateClique eliminates the maximal clique $t.cl_{tcl}.num$ from G.

```
while $t.cl_{tcl}.nx \neq t$ do
    RemoveFromClique($t.cl_{tcl}.nx, t.cl_{tcl}.nx.index$)
FreeClique($t.cl_{tcl}.num$)
$t.cl_{tcl} \leftarrow$ NIL
```

C.2.2 Adding a vertex

Recall the algorithm AddVertex, found in Figure 5.3, which is used to relabel an r-mino when a vertex is added. The following pseudocode can be used to implement AddVertex.

AddVertex(G, X)

Input: An adjacency labelling of an r-mino G created using our dynamic scheme, and a subset X of V_G.

Output: Let G' be the graph formed by adding a new vertex v to G, where v is adjacent to exactly those vertices in X. Providing G' is an r-mino, the output is an adjacency labelling of G'. If G' is not an r-mino, the output indicates as such.

```
$v \leftarrow$ GetIdentifierVertex()
if $X = \emptyset$ then
    $\overline{I} \leftarrow$ NIL
    Push($\overline{I}, v$)
    MakeNewClique($\overline{I}$)
else $(\mathcal{C}, \overline{C}) \leftarrow$ GetCliques($X$)
    $\mathcal{D} \leftarrow \emptyset$
    while $\overline{C} \neq$ NIL do
        $(x, xcl) \leftarrow$ Pop($\overline{C}$)
        $\mathcal{C} \leftarrow \mathcal{C} \setminus \{x.cl_{xcl}.num\}$
        $(C, \overline{C}) \leftarrow$ GetCliqueMembers($x, xcl$)
        $\overline{C'} \leftarrow$ NIL
        $subset \leftarrow 1$
        while $\overline{C} \neq$ NIL do
            $(c, ccl) \leftarrow$ Pop($\overline{C}$)
            if $c \in X$ then
                Push($\overline{C'}, c$)
            else $subset \leftarrow 0$
        if GetCommonCliques($\overline{C'}$) $\not\subseteq \mathcal{C} \cup \mathcal{D}$ then
            $\mathcal{D} \leftarrow \mathcal{D} \cup \{x.cl_{xcl}.num\}$
            if $subset = 1$ then
```

```
22              ADDTOCLIQUE($v, x, xcl$)
23          else PUSH($\overline{C'}, v$)
24              MAKENEWCLIQUE($\overline{C'}$)
```

1: We obtain an identifier for the new vertex.

2-5: If v is an isolated vertex, then it belongs to exactly one maximal clique, $\{v\}$.

6: We determine $\mathcal{C}$, the set of maximal cliques that contain a vertex of X.

7-24: For each maximal clique C in $\mathcal{C}$, we wish to know if $C' \cup \{v\}$ is a maximal clique of G', where $C' = C \cap X$. When we select a member C from $\mathcal{C}$, we discard it from $\mathcal{C}$, however, if $C' \cup \{v\}$ is a maximal clique, then we add C to $\mathcal{D}$.

8-11: We select C from $\mathcal{C}$, remove C from $\mathcal{C}$, and determine the vertices in C.

12-18: We determine $C' = C \cap X$. While doing so, we determine if $C \subseteq X$.

19-24: The clique $C' \cup \{v\}$ is maximal if and only if it is not contained in any maximal clique in $\mathcal{C}$ or $\mathcal{D}$. If $C' \cup \{v\}$ is maximal and $C \subseteq X$, then we add v to C, as C will no longer be maximal in G'. If $C' \cup \{v\}$ is maximal and $C \not\subseteq X$, then we make a new maximal clique of $C' \cup \{v\}$, as C will continue to be maximal in G'.

MAKENEWCLIQUE($\overline{S}$)

Input: A non-empty stack $\overline{S}$ of vertices.

Output: A maximal clique consisting of the vertices in $\overline{S}$.

```
$t \leftarrow$ POP($\overline{S}$)
ADDTOCLIQUE($t, t,$ NIL)
while $\overline{S} \neq$ NIL do
    $s \leftarrow$ POP($\overline{S}$)
    ADDTOCLIQUE($s, t, t.cin$)
```

ADDTOCLIQUE(y, w, wcl)

Input: A triple (y, w, wcl), where w and y are vertices. If $w \neq y$, then $1 \leq wcl \leq w.cin$, otherwise, $wcl =$ NIL.

Output: If $w \neq y$, then ADDTOCLIQUE adds the vertex y to the maximal clique $w.cl_{wcl}.num$. Otherwise, ADDTOCLIQUE initiates a new maximal clique $\{y\}$.

```
$y.cin \leftarrow y.cin + 1$
CHECKRCLIQUES($y$)
$ycl \leftarrow y.cin$
if $w = y$ then
    $cliquenum \leftarrow$ GETIDENTIFIERCLIQUE()
    $y.cl_{ycl}.num \leftarrow cliquenum$
    $y.cl_{ycl}.prev.id \leftarrow y$
    $y.cl_{ycl}.prev.index \leftarrow ycl$
    $y.cl_{ycl}.nx.id \leftarrow y$
    $y.cl_{ycl}.nx.index \leftarrow ycl$
else $y.bicl_{ybicl}.num \leftarrow w.bicl_{wbicl}.num$
```

```
    $z \leftarrow w.cl_{wcl}.nx.id$
    $zcl \leftarrow w.cl_{wcl}.nx.index$
    $w.cl_{wcl}.nx.id \leftarrow y$
    $w.cl_{wcl}.nx.index \leftarrow ycl$
    $y.cl_{ycl}.prev.id \leftarrow w$
    $y.cl_{ycl}.prev.index \leftarrow wcl$
    $y.cl_{ycl}.nx.id \leftarrow z$
    $y.cl_{ycl}.nx.index \leftarrow zcl$
    $z.cl_{zcl}.prev.id \leftarrow y$
    $z.cl_{zcl}.prev.index \leftarrow ycl$
```

CHECKRCLIQUES(t)

Input: A vertex t.

Output: CHECKRCLIQUES ensures that t belongs to no more than r maximal cliques.

```
if $t.cin > r$ then
    error the graph is no longer an $r$-mino
```

C.2.3 Deleting an edge

Recall the algorithm DELETEEDGE, found in Figure 5.4, which is used to relabel an r-mino when an edge is deleted. The following pseudocode can be used to implement DELETEEDGE.

DELETEEDGE(G, u, v)

Input: An adjacency labelling of a r-mino G created using our dynamic scheme, and two distinct vertices u and v of V_G for which $uv \in E_G$.

Output: An adjacency labeling of a graph G' formed by deleting the edge uv from G, providing G' is an r-mino. If G' is not an r-mino, then the output indicates as such.

```
$(\mathcal{C}_u, \overline{\mathcal{C}_u}) \leftarrow$ GETCLIQUES($\{u\}$)
$(\mathcal{C}_v, \overline{\mathcal{C}_v}) \leftarrow$ GETCLIQUES($\{v\}$)
$\overline{\mathcal{C}} \leftarrow$ NIL
while $\overline{\mathcal{C}_v} \neq$ NIL do
    $(x, xcl) \leftarrow$ POP($\overline{\mathcal{C}_v}$)
    if $x.cl_{xcl}.num \in \mathcal{C}_u$ then
        PUSH($\overline{\mathcal{C}}, (x, xcl)$)
while $\overline{\mathcal{C}} \neq$ NIL do
    $(x, xcl) \leftarrow$ POP($\overline{\mathcal{C}}$)
    $(C, \overline{C}) \leftarrow$ GETCLIQUEMEMBERS($x, xcl$)
    $\overline{C'_u} \leftarrow$ NIL
    $\overline{C'_v} \leftarrow$ NIL
    while $\overline{C} \neq$ NIL do
        $(c, ccl) \leftarrow$ POP($\overline{C}$)
        if $c \neq u$ then
            PUSH($\overline{C'_u}, c$)
        else $ucl \leftarrow ccl$
        if $c \neq v$ then
            PUSH($\overline{C'_v}, c$)
        else $vcl \leftarrow ccl$
    if GETCOMMONCLIQUES($\overline{C'_v}$) $= \{x.cl_{xcl}.num\}$ then
        REMOVEFROMCLIQUE($v, vcl$)
        if GETCOMMONCLIQUES($\overline{C'_u}$) $= \{x.cl_{xcl}.num\}$ then
```

```
24              MakeNewClique($\overline{C'_u}$)
25          elseif GetCommonCliques($\overline{C'_u}$) $= \{x.cl_{xcl}.num\}$ then
26              RemoveFromClique($u, ucl$)
27          else EliminateClique($x, xcl$)
```

1,2: We determine $\mathcal{C}_u$ and $\mathcal{C}_v$, the set of maximal cliques that contain u and v, respectively.

3-7: We obtain $\mathcal{C} = \mathcal{C}_u \cap \mathcal{C}_v$.

8-27: For each maximal clique C in $\mathcal{C}$, let $C'_u = C \setminus \{u\}$, and let $C'_v = C \setminus \{v\}$. Since C will no longer be a clique, we wish to know whether C'_u and C'_v are maximal in G'.

8-20: We calculate C'_u and C'_v. The values ucl and vcl are the indices for which $u.cl_{ucl}.num = v.cl_{vcl}.num = x.cl_{xcl}.num$.

21-27: The cliques C'_u and C'_v are maximal if and only if it they are not contained in a maximal clique of G, other than C. If only C'_u is maximal in G', we develop this maximal clique by removing u from C; similarly, if only C'_v is maximal in G', we develop this maximal clique by removing v from C. If neither C'_u nor C'_v is a maximal clique in G', we eliminate the maximal clique C. If both C'_u and C'_v are maximal in G', then we develop C'_v by removing v from C, however, C'_u must be developed by establishing a new maximal clique.

C.2.4 Adding an edge

Recall the algorithm AddEdge, found in Figure 5.5, which is used to relabel an r-mino when an edge is added. The following pseudocode can be used to implement AddEdge.

AddEdge(G, u, v)

Input: An adjacency labelling of an r-mino G created using our dynamic scheme, and two distinct vertices u and v of V_G for which $uv \notin E_G$.

Output: An adjacency labeling of a graph G' formed by adding the edge uv to G, providing G' is an r-mino. If G' is not an r-mino, then the output indicates as such.

```
1   if $v.cl_1.nx = v$ then
2       EliminateClique($v, 1$)
3   $X \leftarrow$ GetNeighbours($v$) $\cup \{u\}$
4   $(\mathcal{C}, \overline{\mathcal{C}}) \leftarrow$ GetCliques($\{u\}$)
5   $\mathcal{D} \leftarrow \emptyset$
6   while $\overline{\mathcal{C}} \neq$ nil do
7       $(x, xcl) \leftarrow$ Pop($\overline{\mathcal{C}}$)
8       $\mathcal{C} \leftarrow \mathcal{C} \setminus \{x.cl_{xcl}.num\}$
9       $(C, \overline{C}) \leftarrow$ GetCliqueMembers($x, xcl$)
10      $\overline{C'} \leftarrow$ nil
11      $subset \leftarrow 1$
12      while $\overline{C} \neq$ nil do
```

```
13          $(c, ccl) \leftarrow$ Pop($\overline{C}$)
14          if $c \in X$ then
15              Push($\overline{C'}, c$)
16          else $subset \leftarrow 0$
17      if GetCommonCliques($\overline{C'}$) $\not\subseteq \mathcal{C} \cup \mathcal{D}$ then
18          $\mathcal{D} \leftarrow \mathcal{D} \cup \{x.cl_{xcl}.num\}$
19          if $subset = 1$ then
20              AddToClique($v, x, xcl$)
21          else Push($\overline{C'}, v$)
22              MakeNewClique($\overline{C'}$)
```

1-2: If v is an isolated vertex in G, then $\{u, v\}$ will be a clique in G'. We eliminate the maximal clique $\{v\}$ now, but will later add v to some maximal clique containing u.

3: We let X be the set of neighbours of v in G'.

4: We determine $\mathcal{C}$, the set of maximal cliques that contain u.

5-22: For each maximal clique C in $\mathcal{C}$, we wish to know if $C' \cup \{v\}$ is a maximal clique of G', where $C' = C \cap X$. When we select a member C from $\mathcal{C}$, we discard it from $\mathcal{C}$, however, if $C' \cup \{v\}$ is a maximal clique, then we add C to $\mathcal{D}$.

6-9: We select C from $\mathcal{C}$, remove C from $\mathcal{C}$, and determine the vertices in C.

10-16: We determine $C' = C \cap X$. While doing so, we determine if $C \subseteq X$

17-22: The clique $C' \cup \{v\}$ is maximal if and only if it is not contained in any maximal clique in $\mathcal{C}$ or $\mathcal{D}$. If $C' \cup \{v\}$ is maximal and $C \subseteq X$, then we add v to C, as C will no longer be maximal in G'. If $C' \cup \{v\}$ is maximal and $C \not\subseteq X$, then we make a new maximal clique of $C' \cup \{v\}$, as C will continue to be maximal in G'.

……

GetNeighbours(t)
Input: A vertex t of G.
Output: The set S of neighbours of t in G.

```
$(W, \overline{W}) \leftarrow$ GetCliques($\{t\}$)
$S \leftarrow \emptyset$
while $\overline{W} \neq$ nil do
    $(w, wcl) \leftarrow$ Pop($\overline{W}$)
    $(Y, \overline{Y}) \leftarrow$ GetCliqueMembers($w, wcl$)
    $S \leftarrow S \cup Y$
return $S$
```

C.3 r-bics

C.3.1 Deleting a vertex

Recall the algorithm DeleteVertex, found in Figure 5.7, which is used to relabel a r-bic when a vertex is deleted. The following pseudocode can be used to implement DeleteVertex.

DeleteVertex(G, v)

Input: An adjacency labelling of an r-bic G created using our dynamic scheme, and a vertex v in V_G. Here we presume that $r \in O(1)$, thereby ensuring error-detection.

Output: An adjacency labeling of an r-bic G' formed by deleting v from G.

```
1   for $i \leftarrow 1$ to $v.bin$ do
2       $(B, \overline{B}) \leftarrow$ GetBicliqueMembers$(v, i)$
3       $\overline{B'} \leftarrow$ NIL
4       while $\overline{B} \neq$ NIL do
5           $(b, bbicl) \leftarrow$ Pop$(\overline{B})$
6           $bpart \leftarrow b.bicl_{bbicl}.part$
7           if $b \neq v$ then
8               Push$(\overline{B'}, (b, bpart))$
9       if GetCommonBicliques$(\overline{B'}) = \{v.bicl_i.num\}$ then
10          RemoveFromBiclique$(v, i)$
11      else EliminateBiclique$(v, i)$
12  FreeVertex$(v)$
```

1-8: For each maximal clique B containing v, we obtain the biclique $B' = B \setminus \{v\}$. The variable *bpart* denotes the part of the partition of B to which b belongs.

9-11: If the members of B' share another maximal biclique besides B, then we eliminate B. Otherwise, we simply remove v from B.

12: We free the identifier of v for future use.

……

GetBicliqueMemebers$(t, tbicl)$

Input: A pair $(t, tbicl)$, where t is a vertex of G and $tbicl$ is an index between 1 and $t.bin$.

Output: Let S denote the set of all vertices in the maximal biclique $t.bicl_{tbicl}.num$. GetBicliqueMembers returns the pair $(S, \overline{S})$, where $\overline{S}$ is a stack consisting of all pairs of the form $(s, sbicl)$, where $s.bicl_{sbicl}.num = t.bicl_{tbicl}.num$.

```
$S \leftarrow \emptyset$
$\overline{S} \leftarrow$ NIL
$s \leftarrow t$
$sbicl \leftarrow tbicl$
Push$(\overline{S}, (s, sbicl))$
$S \leftarrow S \cup \{s\}$
while $s.bicl_{sbicl}.nx.id \neq t$ do
    $y \leftarrow s.bicl_{sbicl}.nx.id$
    $sbicl \leftarrow s.bicl_{sbicl}.nx.index$
    $s \leftarrow y$
    Push$(\overline{S}, (s, sbicl))$
    $S \leftarrow S \cup \{s\}$
return $(S, \overline{S})$
```

……

GetCommonBicliques($\overline{S}$)

Input: A non-empty stack $\overline{S}$ of pairs of the form (s, t), where s is a vertex, and t is a value, either 0 or 1.

Output: For each vertex s in $\overline{S}$, let $\mathcal{B}_s$ denote the set of maximal bicliques containing s. GetCommonBicliques returns $\bigcap_{s \in \overline{S}} \mathcal{B}_s$.

```
$(s, t) \leftarrow$ Pop($\overline{S}$)
$(A, \overline{A}) \leftarrow$ GetBicliques($\{s\}$)
while $\overline{S} \neq$ nil do
    $(s, t) \leftarrow$ Pop($\overline{S}$)
    $(Y, \overline{Y}) \leftarrow$ GetBicliques($\{s\}$)
    $A \leftarrow A \cap Y$
return $A$
```

……

GetBicliques(S)

Input: A non-empty set of vertices S.

Output: Let $\mathcal{B}_s$ denote the set of all maximal bicliques containing a vertex s, and let $\mathcal{B}$ denote $\bigcup_{s \in S} \mathcal{B}_s$. GetBicliques returns the pair $(\mathcal{B}, \overline{T})$, where T is a stack containing an element of the form (t, i), for each entry $t.bicl_i.num$ of $\mathcal{B}$.

```
$T \leftarrow \emptyset$
$\overline{T} \leftarrow$ nil
for $s \in S$ do
    for $i \leftarrow 1$ to $s.bin$ do
        if $s.bicl_i.num \notin T$ then
            $T \leftarrow T \cup \{s.bicl_i.num\}$
            Push($\overline{T}, (s, i)$)
return $(T, \overline{T})$
```

……

RemoveFromBiclique($y, ybicl$)

Input: A pair $(y, ybicl)$, where y is a vertex of G and $ybicl$ is an index between 1 and $y.bin$. Moreover, $y.bicl_{ybicl}.nx.id \neq y$.

Output: RemoveFromBiclique eliminates y from the maximal biclique $y.bicl_{ybicl}.num$.

```
$t \leftarrow y.bicl_{ybicl}.prev.id$
$tbicl \leftarrow y.bicl_{ybicl}.prev.index$
$z \leftarrow y.bicl_{ybicl}.nx.id$
$zbicl \leftarrow y.bicl_{ybicl}.nx.index$
$y.bicl_{ybicl} \leftarrow$ NIL
$t.bicl_{xbicl}.nx.id \leftarrow z$
$t.bicl_{xbicl}.nx.index \leftarrow zbicl$
$z.bicl_{zbicl}.prev.id \leftarrow t$
$z.bicl_{zbicl}.prev.index \leftarrow tbicl$
```

……

EliminateBiclique($t, tbicl$)

Input: A pair $(t, tbicl)$, where t is a vertex of G and $tbicl$ is an index between 1 and $t.bin$.

Output: EliminateBiclique eliminates the maximal biclique $t.bicl_{tbicl}.num$ from G.

```
while $t.bicl_{tbicl}.nx \neq t$ do
    RemoveFromBiclique($t.bicl_{tbicl}.nx, t.bicl_{tbicl}.nx.index$)
```

```
3   FREEBICLIQUE($t.bicl_{tbicl}.num$)
4   $t.bicl_{tbicl} \leftarrow$ NIL
```

C.3.2 Adding a vertex

Recall the algorithm ADDVERTEX, found in Figure 5.8, which is used to relabel an r-bic when a vertex is added. The following pseudocode can be used to implement ADDVERTEX.

ADDVERTEX(G, X)

Input: An adjacency labelling of an r-bic G created using our dynamic scheme, and a subset X of V_G. Here we assume that $X \neq \emptyset$.

Output: Let G' be the graph formed by adding a new vertex v to G, where v is adjacent to exactly those vertices in X. Providing G' is an r-bic, the output is an adjacency labelling of G'. If G' is not an r-bic, the output indicates as such.

```
1   $v \leftarrow$ GETIDENTIFIERVERTEX()
2   $x \leftarrow$ some member of $X$
3   $(\mathcal{B}, \overline{\mathcal{B}}) \leftarrow$ GETALLBICLIQUES($x$)
4   $\mathcal{D} \leftarrow \emptyset$
5   while $\overline{\mathcal{B}} \neq$ NIL do
6       $(c, cbicl) \leftarrow$ POP($\overline{\mathcal{B}}$)
7       $\mathcal{B} \leftarrow \mathcal{B} \setminus \{c.bicl_{cbicl}.num\}$
8       $(B, \overline{B}) \leftarrow$ GETBICLIQUEMEMBERS($c, cbicl$)
9       $\overline{P_0^B} \leftarrow$ NIL
10      $\overline{P_1^B} \leftarrow$ NIL
11      $equal_0 \leftarrow 1$
12      $equal_1 \leftarrow 1$
13      while $\overline{B} \neq$ NIL do
14          $(b, bbicl) \leftarrow$ POP($\overline{B}$)
15          $bpart \leftarrow b.bicl_{bbicl}.part$
16          if $b \in X$ then
17              PUSH($\overline{P_{1-bpart}^B}, (b, bpart)$)
18              $equal_{bpart} \leftarrow 0$
19          else PUSH($\overline{P_{bpart}^B}, (b, bpart)$)
20              $equal_{1-bpart} \leftarrow 0$
21      $include \leftarrow 0$
22      for $i \leftarrow 0$ to $1$ do
23          if GETCOMMONBICLIQUES($\overline{P_i^B}$) $\not\subseteq \mathcal{B} \cup \mathcal{D}$ then
24              $include \leftarrow 1$
25              if $subset = 1$ then
26                  ADDTOBICLIQUE($v, i, c, cbicl$)
27              else PUSH($\overline{P_i^B}, (v, i)$)
28                  MAKENEWBICLIQUE($\overline{P_i^B}$)
29      if $include = 1$ then
30          $\mathcal{D} \leftarrow \mathcal{D} \cup \{c.bicl_{cbicl}.num\}$
```

1: We obtain an identifier for the new vertex.

2,3: We determine $\mathcal{B}$, the set of maximal bicliques of G.

4-30: For each maximal biclique B in $\mathcal{B}$, let $\{B_0, B_1\}$ be the bipartition of B, and let $\{P_0^B, P_1^B\}$ denote the partition of B defined by $b \in P_i^B$ if and only if $b \in X$ and

$b \notin B_i$, or $b \notin X$ and $b \in B_i$. We wish to know if $P_i^B \cup \{v\}$ is a maximal biclique of G'.

When we select a member B from $\mathcal{B}$, we discard it from $\mathcal{B}$, however, if either $P_0^B \cup \{v\}$ or $P_1^B \cup \{v\}$ is a maximal biclique, then we add B to $\mathcal{D}$.

5-8: We select B from $\mathcal{B}$, remove B from $\mathcal{B}$, and determine the vertices in B.

9-20: We determine P_0^B and P_1^B. The variables $equal_0$ and $equal_1$ are used to let us know if $P_0^B = B$ or $P_1^B = B$, respectively.

21-30: The biclique $P_i^B \cup \{v\}$ is maximal if and only if it is not contained in any maximal biclique in $\mathcal{B}$ or $\mathcal{D}$. If $P_i^B \cup \{v\}$ is maximal and $P_i^B = B$, then we add v to B, as B will no longer be maximal in G'. If $P_i^B \cup \{v\}$ is maximal and $P_i^B \neq B$, then we make a new maximal biclique of $P_i^B \cup \{v\}$, as B will continue to be maximal in G'. Specifically, in the new biclique the vertices of P_i^B will belong to the same part of the bipartition that they belonged to in B, while v will be added to the i^{th} part of the bipartition.

29,30: Recall that B is added to $\mathcal{D}$ if either $P_0^B \cup \{v\}$ or $P_1^B \cup \{v\}$ is maximal in G'.

……

GETALLBICLIQUES(t)

Input: A vertex t.

Output: Let $\mathcal{B}_s$ denote the set of all maximal bicliques containing a vertex s, and let $\mathcal{B}$ denote $\bigcup_{s \in V_G} \mathcal{B}_s$. GETALLBICLIQUES returns the pair $(\mathcal{B}, \overline{T})$, where T is a stack containing an element of the form (t, i), for each entry $t.bicl_i.num$ of $\mathcal{B}$.

```
$(S, \overline{S}) \leftarrow$ GETALLVERTICES($\{t\}$)
$T \leftarrow \emptyset$
$\overline{T} \leftarrow$ NIL
while $\overline{S} \neq$ NIL do
    $s \leftarrow$ POP($\overline{S}$)
    for $i \leftarrow 1$ to $s.bin$ do
        if $s.bicl_i.num \notin T$ then
            $T \leftarrow T \cup \{s.bicl_i.num\}$
            PUSH($\overline{T}, (s, i)$)
return $(T, \overline{T})$
```

……

GETALLVERTICES(t)

Input: A vertex t.

Output: The pair $(V_G, \overline{V_G})$.

```
$(Z, \overline{Z}) \leftarrow$ GETBICLIQUES($\{t\}$)
$S \leftarrow \emptyset$
$\overline{S} \leftarrow$ NIL
while $\overline{Z} \neq$ NIL do
    $(z, zbicl) \leftarrow$ POP($\overline{Z}$)
    $(Y, \overline{Y}) \leftarrow$ GETBICLIQUEMEMBERS($z, zbicl$)
    while $\overline{Y} \neq$ NIL do
```

```
8            $(y, ybicl) \leftarrow$ POP($\overline{Y}$)
9            if $y \notin S$ then
10               $S \leftarrow S \cup \{y\}$
11               PUSH($\overline{S}, y$)
12   return $(S, \overline{S})$
```

1-12: Using a single vertex t, we obtain $S = V_G$ by taking the union of the vertices in all the bicliques in which it is contained.

1: We first obtain $\mathcal{B}_t$, the set of all maximal bicliques containing t.

2-6: For each maximal biclique of $\mathcal{B}_t$, we obtain its vertices.

……

ADDTOBICLIQUE($y, ypart, t, tbicl$)

Input: A 4-tuple $(y, ypart, t, tbicl)$, where t and y are vertices and $ypart$ is a value, either 0 or 1. If $t \neq y$, then $1 \leq tbicl \leq t.bin$, otherwise $tbicl =$ NIL.

Output: If $t \neq y$, then ADDTOBICLIQUE adds the vertex y to the $ypart^{\text{th}}$ part of the bipartition of the maximal biclique $t.bicl_{tbicl}.num$. Otherwise, ADDTOBICLIQUE initiates a new maximal biclique $\{y\}$.

```
$y.bin \leftarrow y.bin + 1$
CHECKRBICLIQUES($y$)
$ybicl \leftarrow y.bin$
$y.bicl_{ybicl}.part \leftarrow ypart$
if $t = y$ then
    $bicliquenum \leftarrow$ GETIDENTIFIERBICLIQUE()
    $y.bicl_{ybicl}.num \leftarrow bicliquenum$
    $y.bicl_{ybicl}.prev.id \leftarrow y$
    $y.bicl_{ybicl}.prev.index \leftarrow ybicl$
    $y.bicl_{ybicl}.nx.id \leftarrow y$
    $y.bicl_{ybicl}.nx.index \leftarrow ybicl$
else $y.bicl_{ybicl}.num \leftarrow t.bicl_{tbicl}.num$
    $z \leftarrow t.bicl_{tbicl}.nx.id$
    $zbicl \leftarrow t.bicl_{tbicl}.nx.index$
    $t.bicl_{tbicl}.nx.id \leftarrow y$
    $t.bicl_{tbicl}.nx.index \leftarrow ybicl$
    $y.bicl_{ybicl}.prev.id \leftarrow t$
    $y.bicl_{ybicl}.prev.index \leftarrow tbicl$
    $y.bicl_{ybicl}.nx.id \leftarrow z$
    $y.bicl_{ybicl}.nx.index \leftarrow zbicl$
    $z.bicl_{zbicl}.prev.id \leftarrow y$
    $z.bicl_{zbicl}.prev.index \leftarrow ybicl$
```

……

CHECKRBICLIQUES(t)

Input: A vertex t.

Output: CHECKRBICLIQUES ensures that t belongs to no more than r maximal bicliques.

```
if $t.bin > r$ then
    error the graph is no longer an $r$-bic
```

……

MAKENEWBICLIQUE($\overline{S}$)

Input: A non-empty stack $\overline{S}$ of pairs of the form $(s, spart)$, where s is a vertex and $spart$ is a value, either 0 or 1.

Output: A maximal biclique consisting of the vertices in $\overline{S}$, where a vertex s is placed in the $spart^{\text{th}}$ part of the bipartition.

```
$(t, tpart) \leftarrow$ Pop($\overline{S}$)
AddToBiclique($t, tpart, t,$ nil)
while $\overline{S} \neq$ Nil do
    $(s, spart) \leftarrow$ Pop($S$)
    AddToBiclique($s, spart, t, t.bin$)
```

C.3.3 Deleting an edge

Recall the algorithm DeleteEdge, found in Figure 5.9, which is used to relabel a r-bic when an edge is deleted. The following pseudocode can be used to implement DeleteEdge.

DeleteEdge(G, u, v)

Input: An adjacency labelling of an r-bic G created using our dynamic scheme, and two distinct vertices u and v of V_G for which $uv \in E_G$.

Output: An adjacency labeling of a graph G' formed by deleting the edge uv from G, providing G' is an r-bic. If G' is not an r-bic, then the output indicates as such.

```
$(\mathcal{C}, \overline{\mathcal{C}}) \leftarrow$ GetBicliques($\{u\}$)
$(\mathcal{C}_v, \overline{\mathcal{C}_v}) \leftarrow$ GetBicliques($\{v\}$)
$\mathcal{B} \leftarrow \emptyset$
$\overline{\mathcal{B}} \leftarrow$ nil
while $\overline{\mathcal{C}_v} \neq$ nil do
    $(x, xbicl) \leftarrow$ Pop($\overline{\mathcal{C}_v}$)
    if $x \in \mathcal{C}$ then
        $\mathcal{B} \leftarrow \mathcal{B} \cup \{x\}$
        Push($\overline{\mathcal{B}}, (x, xbicl)$)
while $\overline{\mathcal{B}} \neq$ nil do
    $(x, xbicl) \leftarrow$ Pop($\overline{\mathcal{B}}$)
    $(B, \overline{B}) \leftarrow$ GetBicliqueMembers($x, xbicl$)
    if $B \neq \{u, v\}$ then
        $\overline{B'_u} \leftarrow$ nil
        $\overline{B'_v} \leftarrow$ nil
        while $\overline{B} \neq$ nil do
            $(b, bbicl) \leftarrow$ Pop($\overline{B}$)
            $bpart \leftarrow b.bicl_{bbicl}.part$
            if $b \neq u$ then
                Push($\overline{B'_u}, (b, bpart)$)
            else $ubicl \leftarrow bbicl$
            if $b \neq v$ then
                Push($\overline{B'_v}, (b, bpart)$)
            else $vbicl \leftarrow bbicl$
        if GetCommonBicliques($\overline{B'_v}$) = $\{x.bicl_{xbicl}.num\}$ then
            RemoveFromBiclique($v, vbicl$)
            if GetCommonBicliques($\overline{B'_u}$) = $\{x.bicl_{xbicl}.num\}$ then
                MakeNewBiclique($\overline{B'_u}$)
        elseif GetCommonBiliques($\overline{B'_u}$) = $\{x.bicl_{xbicl}.num\}$ then
            RemoveFromBiclique($u, ubicl$)
        else EliminateBiclique($x, xbicl$)
$(V, \overline{V}) \leftarrow$ GetAllVertices($u$)
$X_u \leftarrow$ GetNeighbours($u$) $\setminus \{v\}$
```

```
34  $X_v \leftarrow$ GetNeighbours$(v) \setminus \{u\}$
35  $W \leftarrow \emptyset$
36  while $\overline{V} \neq$ nil do
37      $x \leftarrow$ Pop$(\overline{V})$
38      if ($x \in X_u$ and $x \in X_v$) or ($x \notin X_u$ and $x \notin X_v$) then
39          $W \leftarrow W \cup \{x\}$
40  $\mathcal{D} \leftarrow \emptyset$
41  while $\overline{\mathcal{C}} \neq$ nil do
42      $(c, cbicl) \leftarrow$ Pop$(\overline{\mathcal{C}})$
43      $\mathcal{C} \leftarrow \mathcal{C} \setminus \{c.bicl_{cbicl}.num\}$
44      $(B, \overline{B}) \leftarrow$ GetBicliqueMembers$(c, cbicl)$
45      $\overline{B'} \leftarrow$ nil
46      $subset \leftarrow 1$
47      $hasv \leftarrow 0$
48      while $\overline{B} \neq$ nil do
49          $(b, bbicl) \leftarrow$ Pop$(\overline{B})$
50          $bpart \leftarrow b.bicl_{bbicl}.part$
51          if $b \in W$ then
52              if $b = u$ then
53                  $upart \leftarrow b.bicl_{bbicl}.part$
54              if $b = v$ then
55                  $upart \leftarrow b.bicl_{bbicl}.part$
56                  $vbicl \leftarrow bbicl$
57                  $hasv \leftarrow 1$
58              else Push$(\overline{B'}, (b, bpart))$
59          else $subset \leftarrow 0$
60      if GetCommonBicliques$(\overline{B'}) \not\subseteq \mathcal{C} \cup \mathcal{D}$ then
61          $\mathcal{D} \leftarrow \mathcal{D} \cup \{c.bicl_{cbicl}.num\}$
62          if $subset = 1$ then
63              if $hasv = 1$ then
64                  $v.bicl_{vbicl}.part = 1 - vpart$
65              else AddToBiclique$(v, upart, c, cbicl)$
66          elseif $hasv = 1$ then
67              EliminateBiclique$(c, cbicl)$
68              Push$(\overline{B'}, (v, upart))$
69              MakeNewBiclique$(\overline{B'})$
70          else Push$(\overline{B'}, (v, upart))$
71              MakeNewBiclique$(\overline{B'})$
```

1-9: Where $\mathcal{B}_x$ denotes the set of all maximal bicliques containing x, we first determine $\mathcal{C} = \mathcal{B}_u$ and $\mathcal{B} = \mathcal{B}_u \cap \mathcal{B}_v$. For now, $\mathcal{B}$ may contain the maximal biclique consiting of only u and v.

10-24: If any maximal biclique in $\mathcal{B}$ consist of only u and v, then we ignore it. Henceforth, for any B in $\mathcal{B}$, we can assume that $B \neq \{u, v\}$

For each maximal biclique B in $\mathcal{B}$, let $B'_u = B \setminus \{u\}$ and let $B'_v = B \setminus \{v\}$. Since B will no longer be a biclique, we wish to know whether B'_u and B'_v are maximal in G'.

14-24: We calculate B'_u and B'_v. The values *ubicl* and *vbicl* are the indices for which $u.bicl_{ubicl}.num = v.bicl_{vbicl}.num = x.bicl_{xbicl}.num$.

25-31: The bicliques B'_u and B'_v are maximal if and only if it they are not contained in a maximal biclique of G, other than B. If only B'_u is maximal in G', we develop this maximal biclique by removing u from B; similarly, if only B'_v is maximal in G', we develop this maximal biclique by removing v from B. If neither B'_u nor B'_v is a maximal biclique in G', we eliminate the maximal biclique B. If both B'_u and B'_v are maximal in G', then we develop B'_v by removing v from B, however, B'_u must be developed by establishing a new maximal biclique.

32-39: Let X_u and X_v denote the neighbourhoods of u and v in G', respectively. We calculate W, the subset of V_G for which $w \in W$ if and only if $w \in X_u \Leftrightarrow w \in X_v$.

40-71: For each maximal biclique B in $\mathcal{C}$, we wish to know if $B' \cup \{v\}$ is a maximal biclique of G', where $B' = (B \setminus \{v\}) \cap W$. When we select a member B from $\mathcal{C}$, we discard it from $\mathcal{C}$, however, if $B' \cup \{v\}$ is a maximal biclique, then we add B to $\mathcal{D}$.

42-44: We select B from $\mathcal{C}$, remove B from $\mathcal{C}$, and determine the vertices in B.

45-59: We determine $B' = (B \setminus \{v\}) \cap W$. The value *vbicl* is the index for which $v.bicl_{vbicl}.num = x.bicl_{xbicl}.num$. The values *upart* and *vpart* are the parts of the biparition of B that u and v belong to, respectively. While determining B', we are also able to determine if $B' \subseteq W$ and if $v \in B$.

60-71: The biclique $B' \cup \{v\}$ is maximal if and only if it is not contained in any maximal biclique in $\mathcal{C}$ or $\mathcal{D}$. If $B \subseteq W$ and $v \in B$, then $B = \{u, v\}$, so we merely switch the value of $v.part$ that corresponds to B. If $B \subseteq W$ and $v \notin B$, then we simply add v to B as B will no longer be maximal in G'. If $B \not\subseteq W$ and $v \in B$, then B no longer remains maximal, so we replace B with $B' \cup \{v\}$. In this case, $B' \cup \{v\} = B \cap W = \{u, v\}$, where u and v will belong to the same parts of the bipartition. Finally, if $B \not\subseteq W$ and $v \notin B$, then we create a new maximal biclique $B' \cup \{v\}$, as B will continue to remain maximal in G'.

GetNeighbours(t)
Input: A vertex t of G.
Output: The set S of neighbours of t in G.

```
$(Z, \overline{Z}) \leftarrow$ GetBicliques($\{t\}$)
$S \leftarrow \emptyset$
while $\overline{Z} \neq$ nil do
    $(z, zcl) \leftarrow$ Pop($\overline{Z}$)
    $(Y, \overline{Y}) \leftarrow$ GetBicliqueMembers($z, zcl$)
    $S \leftarrow S \cup Y$
return $S$
```

C.3.4 Adding an edge

As discussed in Section 5.2.2, the pseudocode required to add an edge would be similar to that presented in Section C.3.3, differing only in the definition of W, and the fact u and v will now need to belong to different parts of any common biclique.

C.4 Proper interval graphs

As we did in Chapter 6, we maintain the convention of offsetting vertex level conditions in square brackets, for example, "$F_L(B(v)) = B(v)$ $[f_L(v) = b(v)]$". As well, recall that nearly all of the conditions mentioned herein can be tested in O(1) time. As such, we will only comment on the time required to check a condition if it takes $\omega(1)$ time to check the condition. Furthermore, recall that, for any vertex v and pointer Q, $Q(P(B(v)))$ can be "followed" in O(1) time, using the labels of v and $P(B(v))$. For simplicity, when referring to the vertex $P(B(v))$, we will use the more compact notation $P(v)$; in turn, when referring to the vertex $Q(P(v))$, we will use the notation $Q(v)$.

When we discussed the relabeller in Chapter 6, we saw several instances where additional criteria had to be tested in order to confirm that G' was a proper interval graph. The reader should consult Chapter 6 to see when such additional criteria must be tested. In the ensuing discussion, we focus on the specific actions of the relabeller when we know that G' is a proper interval graph.

C.4.1 Deleting a vertex

Let v be the vertex to be deleted, where X denotes the neighbourhood of v in G. As well, let the contig containing $B(v)$ be $B_1 \preceq \ldots \preceq B_i \preceq \ldots \preceq B_l \preceq \ldots \preceq B_j \preceq \ldots \preceq B_k$, where $B_l = B(v)$, $B_i = F_L(B_l)$, and $B_j = F_R(B_l)$. The action of the relabeller depends on whether $B_l = \{v\}$ $[nx(v) = v]$.

If B_l contains another vertex besides v, then the straight enumeration remains the same, however, v is removed from B_l. Specifically, our labelling is amended as follows.

- Remove all references to v.
    - We must change all references to v as a pointer vertex. Specifically, if $v = P(v)$, then we make $nx(v)$ the pointer vertex by changing its label to reflect the pointers, and changing the labels of all the vertices in B_l to reflect that $nx(v)$ is the new pointer vertex. This change can be done in $O(|B_l|) \in O(|X|)$ time by traversing B_l, beginning at v. Let q be the resulting pointer vertex of B_l.
    - We must change all references to v in I_L and I_R pointers. Providing $I_L(B_l) \neq \text{NIL}$ $[I_L(q) \neq \text{NIL}]$, set $I_R(I_L(q))$ to q. Similarly, providing $I_R(B_l) \neq \text{NIL}$, set $I_L(I_R(q))$ to q. These changes take O(1) time.

  - We must change all references to v in F_L and F_R pointers. Specifically, for any block B, if $F_L(P(B))$ or $F_R(P(B))$ is v, then we change its value to q. Now, if $F_R(P(B)) = v$, then, by Lemma 6.7 (umbrella property), $B_i \preceq B \preceq B_l$; similarly, if $F_L(P(B)) = v$, then $B_l \preceq B \preceq B_j$. As such, we can recursively follow I_L and I_R pointers to determine all such blocks B. These changes take $O(deg(B_l)) \in O(|X|)$ time.
  - We must remove v from the circular doubly linked list of the vertices in B_l. This removal takes $O(1)$ time.
- Decrease the value of $s(B_l)$ [$s(q)$] by one. This operation takes $O(1)$ time.
- Delete v. This deletion takes $O(1)$ time.

If v is in a block by itself, we again remove all references to v, however, we may also have to merge blocks, as depicted in Figure 6.4. Specifically, if v is in a block by itself, our labelling is changed as follows.

- If $1 < i < l$ [$f_L(v) \neq f_L(F_L(v))$ and $f_L(v) \neq b(v)$], $F_L(B_{i-1}) = F_L(B_i)$ [$f_L(I_L(F_L(v))) = f_L(F_L(v))$], and $F_R(B_{i-1}) = B_{l-1}$ [$f_R(I_L(F_L(v))) = b(I_L(v))$], then merge B_i into B_{i-1}.
  - Add the value of $s(B_i)$ to $s(B_{i-1})$ [add $s(P(F_L(v)))$ to $s(P(I_L(F_L(v))))$]. This operation takes $O(1)$ time.
  - Set $I_R(B_{i-1})$ to B_{i+1} [$I_R(I_L(F_L(v)))$ to $I_R(F_L(v))$] and $I_L(B_{i+1})$ to B_{i-1} [$I_L(I_R(F_L(v)))$ to $I_L(F_L(v))$]. These assignments take $O(1)$ time.
  - Update the labels of the vertices of B_i to reflect the fact that $P(B_{i-1})$ [$P(I_L(F_L(v)))$] is the pointer vertex of the merged block. This update can be done in $O(|B_i|) \in O(|X|)$ time by traversing B_i, beginning at $F_L(v)$.
  - Merge the two circular doubly linked lists, using $P(B_{i-1})$ [$P(I_L(F_L(v)))$] and $P(B_i)$ [$P(F_L(v))$] as reference points. This merge takes $O(1)$ time.
- If $l < j < k$, $F_R(B_j) = F_R(B_{j+1})$, and $F_L(B_{j+1}) = B_{l+1}$ then merge B_j into B_{j+1}. This merge takes $O(|B_j|) \in O(|X|)$ time.
- Providing $I_L(B_l) \neq$ NIL [$I_L(v) \neq$ NIL], set $I_R(I_L(B_l))$ to $I_R(B_l)$ [$I_R(I_L(v))$ to $I_R(v)$]. Similarly, providing $I_R(B_l) \neq$ NIL, set $I_L(I_R(B_l))$ to $I_L(B_l)$. These assignments take $O(1)$ time.
- For each block B in $\{B_i, \ldots, B_{l-1}\}$, if $F_R(B) = B_l$ [$f_R(P(B)) = b(v)$], then set $F_R(B)$ to B_{l-1} [$F_R(P(B))$ to $I_L(v)$]. As well, for each block B in $\{B_{l+1}, \ldots, B_j\}$, if $F_L(B) = B_l$, then set $F_L(B)$ to B_{l+1}. These assignments can be done in $O(deg(B_l)) \in O(|X|)$ time, by recursively following I_L and I_R pointers to determine all such blocks B.

- Delete v. This deletion takes O(1) time.
- Relabel the blocks and adjust the b, f_L, and f_R values. This operation takes $O(n)$ time.

C.4.2 Adding a vertex

Let v be the vertex to be added, where X denotes the neighbourhood of v in G'. Hereafter, we assume that Lemmas 6.12 and 6.13 are satisfied by our vertex addition. Recall that, while verifying that Lemmas 6.12 and 6.13 are satisfied, we learn a great deal about the structure of the blocks. This information can be used to help us relabel the vertices.

In describing the relabelling, let us first consider when the members of X belong to one component, C. As in the hypothesis of Lemma 6.13, let $\{B_1, \ldots, B_k\}$ denote the set of blocks in C that are adjacent to v, such that in the contig of C, $B_1 \prec \ldots \prec B_k$. We consider three cases, depending on the value of k.

1. $\underline{k = 1.}$ By Lemma 6.13, B_1 is an end block. Without loss of generality, assume that $B_1 \prec B$, for any block B in C.

   If v is fully adjacent to B_1, and $C = B_1$ ($[f_R(v_1) = b(v_1)]$, then we add v to B_1. Specifically, we do the following.

   - Add v to the circular doubly linked list of vertices in B_1, using v_1 as a reference point. This addition takes O(1) time.
   - Establish the label of v to reflect the pointer vertex for B_1, $P(v_1)$, while setting its b, f_L, and f_R values to those of v_1. Establishing the label of v takes O(1) time.
   - Increase the value of $s(B_1)$ $[s(P(v_1))]$ by one. This adjustment requires O(1) time.

   If v is fully adjacent to B_1, but $C \neq B_1$, then we add the block $B_a = \{v\}$ immediately before B_1. Specifically, we do the following.

   - Establish the trivial circular doubly linked list for B_a. Establishing this circular doubly linked list takes O(1) time.
   - Establish v as the pointer vertex of B_a. For now, assign the pointer values of $P(v_1)$ to v. Establishing this pointer vertex takes O(1) time.
   - Providing $I_L(B_1) \neq$ NIL $[I_L(v_1) \neq$ NIL$]$, set $I_R(I_L(B_1))$ to B_a $[I_R(I_L(v_1))$ to $v]$. This assignment takes O(1) time.

     Once this I_R pointer has been assigned, set $I_R(B_a)$ to B_1 $[I_R(v)$ to $v_1]$ and $I_L(B_1)$ to B_a $[I_L(v_1)$ to $v]$. These assignments take O(1) time.

- Set $F_L(B_a)$ to B_a $[F_L(v)$ to $v]$, $F_R(B_a)$ to B_b $[F_R(v)$ to $v_1]$, and $F_L(B_1)$ to B_a $[F_L(v_1)$ to $v]$. These assignments take O(1) time.
- Set $s(B_a)$ $[s(v)]$ to one. This assignment takes O(1) time.
- Relabel the blocks and adjust the b, f_L, and f_R values. This operation can be done in $\mathrm{O}(n)$ time.

If v is not fully adjacent to B_1, then we partition $B_1 \cup \{v\}$ into $B_a \prec B_b \prec B_c$, where $B_a = \{v\}$, $B_b = X$, and $B_c = B_1 \setminus X$. Specifically, we do the following.

- Establish the trivial circular doubly linked list for B_a. Establishing this circular doubly linked list takes O(1) time.
- Establish v as the pointer vertex of B_a. For now, assign the pointer values of $P(v_1)$ to v. Establishing this pointer vertex takes O(1) time.
- Remove vertices in X from the circular doubly linked list of vertices in B_1 to produce the circular doubly linked list of vertices in B_b and B_c. While doing so, make note of one vertex q_c from B_c. Producing these circular doubly linked lists takes $\mathrm{O}(|B_b|) \in \mathrm{O}(|X|)$ time.
- If $P(v_1) \in X$, then we establish q_c as the new pointer vertex for B_c. Otherwise, if $P(B_1) \notin X$, then we establish v_1 as the new pointer vertex for B_b. Let q_b and q_c be the resulting pointer vertices of B_b and B_c, respectively. For now, assign the pointer values of $P(v_1)$ to q_b and q_c. Establishing these new pointer vertex takes $\mathrm{O}(|B_b| + |B_c|) \in \mathrm{O}(n)$ time.
- Providing $I_L(B_1) \neq \textsc{nil}$ $[I_L(q_c) \neq \textsc{nil}]$, set $I_R(I_L(B_1))$ to B_a $[I_R(I_L(q_c))$ to $v]$. Similarly, providing $I_R(B_1) \neq \textsc{nil}$ $[I_R(q_c) \neq \textsc{nil}]$, set $I_L(I_R(B_1))$ to B_c $[I_L(I_R(q_c))$ to $q_c]$. These assignments take O(1) time.

  Once the above I_l and I_R pointers have been assigned, set $I_R(B_a)$ to B_b $[I_R(v)$ to $q_b]$, $I_L(B_b)$ to B_a $[I_L(q_b)$ to $v]$, $I_R(B_b)$ to B_c $[I_R(q_b)$ to $q_c]$, and $I_L(B_c)$ to B_b $[I_L(q_c)$ to $q_b]$. These assignments also take O(1) time.
- Set $F_L(B_a)$ to B_a $[F_L(v)$ to $v]$, $F_R(B_a)$ to B_b $[F_R(v)$ to $q_b]$, $F_L(B_b)$ to B_a $[F_L(q_b)$ to $v]$, and $F_L(B_c)$ to B_b $[F_L(q_c)$ to $q_b]$. These assignments take O(1) time.
- For each block B in $\{I_R(B_c), \ldots, F_R(B_c)\}$, set $F_L(B)$ to B_b $[F_L(P(B))$ to $q_b]$. These assignments can be made in $\mathrm{O}(deg(B_1))$ time, which could be as large as $\mathrm{O}(n)$ time, by traversing I_R pointers.
- Set $s(B_a)$ $[s(v)]$ to one, $s(B_b)$ $[s(q_b)]$ to $|X|$, and $s(B_c)$ to $s(B_1) - |X|$ [subtract $|X|$ from $s(q_c)$]. These assignments take O(1) time.
- Relabel the blocks and adjust the b, f_L, and f_R values. This operation can be done in $\mathrm{O}(n)$ time.

2. $\underline{k = 2}$. By condition 4 of Lemma 6.13, v must be fully adjacent to at least one of B_1 and B_2. Without loss of generality, assume that v is fully adjacent to B_1. Let $B_i = F_L(B_1)$ and $B_j = F_R(B_1)$.

   Recall that we resolved the relabeller into four cases: adding v to B_1 (or B_2), adding the block $\{v\}$ immediately before B_1, adding the block $\{v\}$ between B_1 and B_2, and partitioning $B_1 \cup B_2 \cup \{v\}$ into $\{v\} \prec B_1 \prec B_2 \cap X \prec B_2 \setminus X$. This resolution took O(1) time.

   If v is added to B_1, then we do the following.

   - Add v to the circular doubly linked list of vertices in B_1, using v_1 as a reference point. This addition takes O(1) time.
   - Establish the label of v to reflect the pointer vertex for B_1, $P(v_1)$, while setting its b, f_L, and f_R values to those of v_1. Establishing the label of v can be done in O(1) time.
   - Increase the value of $s(B_1)$ $[s(P(v_1))]$ by one. This takes O(1) time.

   If the block $B_a = \{v\}$ is added immediately before B_1 in its contig, then we do the following.

   - Establish the trivial circular doubly linked list for B_a. Establishing this circular doubly linked list takes O(1) time.
   - Establish v as the pointer vertex of B_a. For now, assign the pointer values of $P(v_1)$ to v. Establishing this pointer vertex takes O(1) time.
   - Providing $I_L(B_1) \neq \text{NIL}$ $[I_L(v_1) \neq \text{NIL}]$, set $I_R(I_L(B_1))$ to B_a $[I_R(I_L(v_1))$ to $v]$. This assignment takes O(1) time.

     Once this I_L pointer has been assigned, set $I_R(B_a)$ to B_1 $[I_R(v)$ to $v_1]$ and $I_L(B_1)$ to B_a $[I_L(v_1)$ to $v]$. These assignments take O(1) time.
   - Set $F_L(B_a)$ to B_a $[F_L(v)$ to $v]$, $F_R(B_a)$ to B_2 $[F_R(v)$ to $v_2]$, $F_L(B_1)$ to B_a $[F_L(v_1)$ to $v]$, and $F_L(B_2)$ to B_a $[F_L(v_2)$ to $v]$. These assignments take O(1) time.
   - Set $s(B_a)$ $[s(v)]$ to one. This assignment takes O(1) time.
   - Relabel the blocks and adjust the b, f_L, and f_R values. This operation can be done in O(n) time.

   If $B_1 \cup B_2 \cup \{v\}$ is partitioned into $B_a \prec B_1 \prec B_b \prec B_c$, where $B_a = \{v\}$, $B_b = B_2 \cap X$ and $B_c = B_2 \setminus X$, we do the following.

   - Establish the trivial circular doubly linked list for B_a. Establishing this circular doubly linked list takes O(1) time.

- Establish v as the pointer vertex of B_a. For now, assign the pointer values of $P(v_1)$ to v. Establishing this pointer vertex takes O(1) time.
- Remove vertices in X from the circular doubly linked list of vertices in B_2 to produce the circular doubly linked list of vertices in B_b and B_c. While doing so, make note of one vertex q_c from B_c. Producing these circular doubly linked lists takes $O(|B_b|) \in O(|X|)$ time.
- If $P(v_2) \in X$, then we establish q_c as the new pointer vertex for B_c. Otherwise, if $P(v_2) \notin X$, then we establish v_2 as the new pointer vertex for B_b. Let q_b and q_c be the resulting pointer vertices of B_b and B_c, respectively. For now, assign the pointer values of $P(v_2)$ to q_b and q_c. Establishing this new pointer vertex takes $O(|B_b| + |B_c|) \in O(n)$ time.
- Providing $I_L(B_1) \neq \text{NIL}$ $[I_L(v_1) \neq \text{NIL}]$, set $I_R(I_L(B_1))$ to B_a $[I_R(I_L(v_1))$ to $v]$. As well, set $I_L(I_R(B_2))$ to B_c $[I_L(I_R(q_c))$ to $q_c]$. These assignments take O(1) time.

  Once the above I_L and I_R pointers have been assigned, set $I_R(B_a)$ to B_1 $[I_R(v)$ to $v_1]$, $I_L(B_1)$ to B_a $[I_L(v_1)$ to $v]$, $I_R(B_1)$ to B_b $[I_R(v_1)$ to $q_b]$, $I_R(B_b)$ to B_c $[I_R(q_b)$ to $q_c]$, and $I_L(B_c)$ to B_b $[I_L(q_c)$ to $q_b]$. These assignments also take O(1) time.
- Set $F_L(B_a)$ to B_a $[F_L(v)$ to $v]$, $F_R(B_a)$ to B_b $[F_R(v)$ to $q_b]$, $F_L(B_1)$ to B_a $[F_L(v_1)$ to $v]$, and $F_L(B_b)$ to B_a $[F_L(q_b)$ to $v]$. As well, if $F_R(B_1) = B_b$ $[P(F_R(v_1)) = q_b]$, then set $F_R(B_1)$ to B_c $[F_R(v_1)$ to $q_c]$. These assignments take O(1) time.
- For each block B in $\{I_R(B_c), \ldots, F_R(B_c)\}$, if $F_L(B) = B_c$ $[f_L(P(B)) = b(v_2)]$, then set $F_L(B)$ to B_b $[F_L(P(B))$ to $q_b]$. These assignments can be made in $O(deg(B_2))$ time, which could be as large as $\Theta(n)$ time, by traversing I_R pointers.
- Set $s(B_a)$ $[s(v)]$ to one, $s(B_b)$ $[s(q_b)]$ to $|B_2 \cap X|$, and $s(B_c)$ to $s(B_2) - |B_2 \cap X|$ [subtract $s(q_b)$ from $s(q_c)$]. These assignments take O(1) time.
- Relabel the blocks and adjust the b, f_L, and f_R values. This operation can be done in $O(n)$ time.

If the block $B_a = \{v\}$ is added between B_1 and B_2 we do the following.

- Establish the trivial circular doubly linked list for B_a. Establishing this circular doubly linked list takes O(1) time.
- Establish v as the pointer vertex of B_a. For now, assign the pointer values of $P(v_2)$ to v. Establishing this pointer vertex takes O(1) time.
- Set $I_R(B_1)$ to B_a $[I_R(v_1)$ to $v]$, $I_L(B_a)$ to B_1 $[I_L(v)$ to $v_1]$, $I_R(B_a)$ to B_2 $[I_R(v)$ to $v_2]$, and $I_L(B_2)$ to B_a $[I_L(v_2)$ to $v]$. These assignments take O(1) time.

  - Set $F_R(B_a)$ to B_2 [$F_R(v)$ to v_2]. This assignment takes O(1) time.
  - Set $s(B_a)$ [$s(v)$] to one. This assignment takes O(1) time.
  - Relabel the blocks and adjust the b, f_L, and f_R values. This operation can be done in O(n) time.

3. $k \geq 3$. By Lemma 6.13, v is fully adjacent to $B_2, \ldots, B_{k-1}$. Let $B_\alpha = F_R(B_1)$ and let $B_\beta = F_L(B_k)$. As well, let b_α be some vertex in B_α and let b_β be some vertex in B_β

   (a) v is fully adjacent to B_k, but not B_1. To create the new contig, we partition B_1 into $B_a \prec B_b$, where $B_a = B_1 \setminus X$ and $B_b = B_1 \cap X$, and insert the block $B_c = \{v\}$ immediately after B_α. Specifically, we do the following.

      - Establish the trivial circular doubly linked list for B_c. Establishing this circular doubly linked list takes O(1) time.
      - Establish v as the pointer vertex of B_c. For now, assign the pointer values of $P(b_\alpha)$ to v. Establishing this pointer vertex takes O(1) time.
      - Remove vertices in X from the circular doubly linked list of vertices in B_1 to produce the circular doubly linked list of vertices in B_a and B_b. While doing so, make note of one vertex q_a from B_a. Producing these circular doubly linked lists takes O($|X|$) time.
      - If $P(v_1) \in X$, then we establish q_a as the new pointer vertex for B_a. Otherwise, if $P(v_1) \notin X$, then we establish v_1 as the new pointer vertex for B_b. Let q_a and q_b be the resulting pointer vertices of B_a and B_b, respectively. For now, assign the pointer values of $P(v_1)$ to q_a and q_b. Establishing this new pointer vertex takes O($|B_a| + |B_b|$) $\in$ O(n) time.
      - Providing $I_L(B_1) \neq \text{NIL}$ [$I_L(q_a) \neq \text{NIL}$], set $I_R(I_L(B_1))$ to B_a [$I_R(I_L(q_a))$ to q_a]. Similarly, providing $I_R(B_\alpha) \neq \text{NIL}$ [$I_R(b_\alpha) \neq \text{NIL}$], set $I_L(I_R(B_\alpha))$ to B_c [$I_L(I_R(b_\alpha))$ to v]. These assignments take O(1) time.

        Once the above I_L and I_R pointers have been assigned, set $I_R(B_a)$ to B_b [$I_R(q_a)$ to q_b], $I_L(B_b)$ to B_a [$I_L(q_b)$ to q_a], $I_L(B_2)$ to B_b [$I_L(v_2)$ to q_b], $I_R(B_\alpha)$ to B_c [$I_R(b_\alpha)$ to v], and $I_L(B_c)$ to B_α [$I_L(v)$ to b_α]. These assignments also take O(1) time.
      - For each block B in $\{F_L(B_a), \ldots, I_L(B_a)\}$, if $F_R(B) = B_1$ [$f_R(P(B)) = b(v_1)$], then set $F_R(B)$ to B_b [$F_L(P(B))$ to q_b]. These assignments can be made in O($deg(B_1)$) $\in$ O(n) time, by traversing I_L pointers.
      - For each block B in $\{B_b, \ldots, B_\alpha\}$, if $F_L(B) = B_1$ [$f_L(P(B)) = b(v_1)$], then set $F_L(B)$ to B_a [$F_L(P(B))$ to q_a]. These assignments can be made in O($|X|$) time, by traversing I_R pointers.

- For each block B in $\{B_b, \ldots, B_\alpha\}$, if $F_R(B) = B_\alpha$ $[f_R(P(B)) = b(b_\alpha)]$, then set $F_R(B)$ to B_c $[F_R(P(B))$ to $v]$. These assignments can be made in $O(|X|)$ time, by traversing I_R pointers.
- Set $F_L(B_c)$ to B_b $[F_R(v)$ to $q_b]$. As well, if $B_\alpha = B_k$ $[b(F_R(q_a)) = b(v_k)]$, then set $F_R(B_c)$ to B_c $[F_R(v)$ to $v]$, otherwise, $B_\alpha \prec B_k$, so set $F_R(B_c)$ to B_k $[F_R(v)$ to $v_k]$. These assignments take $O(1)$ time.
- For each block B in $\{I_R(B_c), \ldots, B_k\}$, if $F_L(B) = I_R(B_c)$ $[b(I_L(F_L(P(B)))) = b(b_\alpha)]$, then set $F_L(B)$ to B_c $[F_L(P(B))$ to $v]$. These assignments can be made in $O(|X|)$ time, by traversing I_R pointers.
- Set $s(B_c)$ $[s(v)]$ to one, $s(B_b)$ $[s(q_b)]$ to $|B_1 \cap X|$, and $s(B_a)$ to $s(B_1) - |B_1 \cap X|$ [subtract $s(q_b)$ from $s(q_a)$]. These assignments take $O(1)$ time.
- Relabel the blocks and adjust the b, f_L, and f_R values. This operation can be done in $O(n)$ time.

(b) v is fully adjacent to B_1, but not B_k. To create the new contig we partition B_k into $B_d \prec B_e$, where $B_d = B_k \cap X$ and $B_e = B_k \setminus X$, and insert the block $B_c = \{v\}$ immediately before B_β. This scenario is virtually identical to the case when v was fully adjacent to B_k but not B_1.

(c) v is fully adjacent to neither B_1 nor B_k. In essence, this scenario requires the 'combination' of the two previous relabellings. That is, we partition B_1 into $B_a \prec B_b$, where $B_a = B_1 \setminus X$ and $B_b = B_1 \cap X$, we partition B_k into $B_d \prec B_e$, where $B_d = B_k \cap X$ and $B_e = B_k \setminus X$, and we insert $B_c = \{v\}$ between B_α and B_β. As the combination of the previous two relabellings we can obtain the labelling of G' in $O(n)$ time.

(d) v is fully adjacent to both B_1 and B_k. We consider three further cases.

  i. $B_\alpha \prec B_\beta$ $[b(b_\alpha) < b(b_\beta)]$. We add the block $B_a = \{v\}$ between B_α and B_β. Specifically, we do the following.

  - Establish [illegible] trivial circular doubly linked list for B_a. Establishing this circular doubly linked list takes $O(1)$ time.
  - Establish v as the pointer vertex of B_a. For now, assign the pointer values of $P(b_\alpha)$ to v. Establishing this pointer vertex takes $O(1)$ time.
  - Set $I_R(B_\alpha)$ to B_a $[I_R(b_\alpha)$ to $v]$, $I_L(B_a)$ to B_α $[I_L(v)$ to $b_\alpha]$, and $I_L(B_\beta)$ to B_a $[I_L(b_\alpha)$ to $v]$. These assignments take $O(1)$ time.
  - Set $F_L(B_a)$ to B_1 $[F_L(v)$ to $v_1]$, $F_R(B_a)$ to B_k $[F_R(v)$ to $v_k]$. These assignments take $O(1)$ time.
  - For each block B in $\{B_1, \ldots, B_\alpha\}$, if $F_R(B) = B_\alpha$ $[f_R(P(B)) = b(b_\alpha)]$, then set $F_R(B)$ to B_a $[F_R(P(B))$ to $v]$. These assignments can be done

in $O(|X|)$ time by following I_R pointers.

    - For each block B in $\{B_\beta, \ldots, B_k\}$, if $F_L(B) = B_\beta$ [$f_L(P(B)) = b(b_\beta)$], then set $F_L(B)$ to B_a [$F_L(P(B))$ to v]. These assignments can be done in $O(|X|)$ time by following I_L pointers.
    - Set $s(B_a)$ [$s(v)$] to one. This assignment takes $O(1)$ time.
    - Relabel the blocks and adjust the b, f_L, and f_R values. This operation can be done in $O(n)$ time.

ii. $B_\alpha = B_\beta$ [$b(b_\alpha) = b(b_\beta)$]. We consider four cases.

- $F_L(B_\alpha) = B_1$ [$f_L(b_\alpha) = b(v_1)$] and $F_R(B_\alpha) = B_k$ [$f_R(b_\alpha) = b(v_k)$]. Since B_α has the same adjacency as v, we add v to B_α. Specifically, we do the following.
    - Add v to the circular doubly linked list of vertices in B_α using b_α as a reference point. This addition takes $O(1)$ time.
    - Establish the label of v to reflect the pointer vertex for B_α, $P(b_\alpha)$, while setting its b, f_L, and f_R values to those of b_α. Establishing the label of v can be done in $O(1)$ time.
    - Increase the value of $s(B_\alpha)$ [$s(P(b_\alpha))$] by one. This takes $O(1)$ time.
- $F_L(B_\alpha) \prec B_1$ [$f_L(b_\alpha) < b(v_1)$] and $F_R(B_\alpha) = B_k$ [$f_R(b_\alpha) = b(v_k)$]. In this case, we must insert the block $B_a = \{v\}$ immediately after B_α. Specifically, we do the following.
    - Establish the trivial circular doubly linked list for B_a. Establishing this circular doubly linked list takes $O(1)$ time.
    - Establish v as the pointer vertex of B_a. For now, assign the pointer values of $P(b_\alpha)$ to v. Establishing this pointer vertex takes $O(1)$ time.
    - Set $I_L(I_R(B_\alpha))$ to B_a [$I_L(I_R(b_\alpha))$ to v]. This assignment takes $O(1)$ time.

      Once this I_L pointer has been assigned, set $I_R(B_\alpha)$ to B_a [$I_R(b_\alpha)$ to v] and $I_L(B_a)$ to B_α [$I_L(v)$ to b_α]. These assignments also take $O(1)$ time.
    - Set $F_L(B_a)$ to B_1 [$F_L(v)$ to v_1], $F_R(B_a)$ to B_k [$F_R(v)$ to v_k]. These assignments take $O(1)$ time.
    - For each block B in $\{B_1, \ldots, B_\alpha\}$, if $F_R(B) = B_\alpha$ [$f_R(P(B)) = b(b_\alpha)$], then set $F_R(B)$ to B_a [$F_R(P(B))$ to v]. These assignments can be done in $O(|X|)$ time by following I_R pointers.
    - Set $s(B_a)$ [$s(v)$] to one. This assignment takes $O(1)$ time.

    - Relabel the blocks and adjust the b, f_L, and f_R values. This operation can be done in $O(n)$ time.

  - $F_L(B_\alpha) = B_1$ $[f_L(b_\alpha) = b(v_1)]$ and $B_k \prec F_R(B_\alpha)$ $[b(v_k) < f_R(b_\alpha)]$. In this case, the block $B_a = \{v\}$ must be inserted immediately before B_α.
  - $F_L(B_\alpha) \prec B_1$ $[f_L(b_\alpha) < b(v_1)]$ and $B_k \prec F_R(B_\alpha)$ $[b(v_k) < f_R(b_\alpha)]$. Recall that G' is not a proper interval graph.

iii. $B_\beta \prec B_\alpha$ $[b(b_\beta) < b(b_\alpha)]$. By definition, $F_L(B_\alpha) \preceq B_1$ and $B_k \preceq F_R(B_\beta)$. Therefore, $F_L(B_\beta) \preceq B_1$ and $B_k \preceq F_R(B_\alpha)$. We consider four cases.

  - $F_L(B_\beta) = B_1$ $[f_L(b_\beta) = b(v_1)]$ and $F_R(B_\alpha) = B_k$ $[f_R(b_\alpha) = b(v_k)]$. Recall that this case cannot occur.
  - $F_L(B_\beta) \prec B_1$ $[f_L(b_\beta) < b(v_1)]$ and $F_R(B_\alpha) = B_k$ $[f_R(b_\alpha) = b(v_k)]$. Now if $F_L(B_\alpha) = B_1$ $[f_L(b_\alpha) = b(v_1)]$, then B_α has the same adjacency as v so we add v to B_α; adding v to B_α is done in exactly the same manner as when $B_\alpha = B_\beta$, $F_L(B_\alpha) = B_1$, and $F_R(B_\alpha) = B_k$. Otherwise, if $F_L(B_\alpha) \prec B_1$ $[f_L(b_\alpha) < b(v_1)]$, we must insert the block $B_a = \{v\}$ immediately after B_α as $B_1 \prec F_L(I_R(B_\alpha))$. This insertion of the block B_a is done in exactly the same manner as when $B_\alpha = B_\beta$, $F_L(B_\alpha) \prec B_1$, and $F_R(B_\alpha) = B_k$.
  - $F_L(B_\beta) = B_1$ $[f_L(b_\beta) = b(v_1)]$ and $B_k \prec F_R(B_\alpha)$ $[b(v_k) < f_R(b_\alpha)]$. This case is virtually identical to the previous one. If $F_R(B_\beta) = B_k$ $[f_R(b_\beta) = b(v_k)]$, then we add v to B_β. Otherwise, if $B_k \prec F_R(B_\beta)$ $[b(v_k) < f_R(b_\beta)]$, we must insert the block $B_a = \{v\}$ immediately before B_β.
  - $F_L(B_\beta) \prec B_1$ $[f_L(b_\beta) < b(v_1)]$ and $B_k \prec F_R(B_\alpha)$ $[b(v_k) < f_R(b_\alpha)]$. If there exists a block B, such that $F_R(I_L(B_1)) \prec B \prec F_L(I_R(B_k))$, then we must add v to B. We have previously seen how to add v to an existing block. If there does not exist a block B such that $F_R(I_L(B_1)) \prec B \prec F_L(I_R(B_k))$, then we must add the block $B_a = \{v\}$ between $F_R(I_L(B_1))$ and $F_L(I_R(B_k))$. This can be done in a manner similar to the placement of $B_a = \{v\}$ between B_α and B_β in the case where $B_\alpha \prec B_\beta$.

Now let us consider when the members of X belong to two distinct components. Let the contigs of the two components be $\Phi = B_1 \prec \ldots \prec B_k$ and $\Psi = B'_1 \prec \ldots \prec B'_l$, where, without loss of generality, $\Phi \prec \Psi$. As well, let B_i be the leftmost block in Φ to which v is adjacent, and let B'_j be the rightmost block in Ψ to which v is adjacent.

To demonstrate the action of the relabeller, we consider the scenario in which the end blocks to which v is fully adjacent are B_k and B'_1, with v is fully adjacent to B'_j but not B_i. Let B_a be the split block $B_i \setminus X$, B_b be the split block $B_i \cap X$, and B_c be the block $\{v\}$.

The labelling is changed as follows.

- If $I_R(B_k) \neq B'_1$ [$b(I_R(v_k)) = b(v'_1)$], then move Φ over to Ψ.
  - Set $I_R(I_L(B'_1))$ to $I_R(B'_l)$ [$I_R(I_L(v'_1))$ to $I_R(v'_l)$] and $I_R(B'_l)$ to $I_R(B_k)$ [$I_R(v'_l)$ to $I_R(v_k)$]. As well, providing $I_R(B'_l) \neq$ NIL [$I_R(v'_l) \neq$ NIL], set $I_L(I_R(B'_l))$ to $I_L(B'_1)$ [$I_L(I_R(v'_l))$ to $I_L(v'_1)$]. These assignments can be made in O(1) time.

    Once the above I_L and I_R pointers have been assigned, set $I_R(B_k)$ to B'_1 [$I_R(v_k)$ to v'_1], and $I_L(B'_1)$ to B_k [$I_L(v'_1)$ to v_k]. These assignments also take O(1) time.
  - Relabel the blocks and adjust the b, f_L, and f_R values. This operation can be done in O(n) time.
- Insert B_c and split B_i to reflect the straight enumeration of G'.
  - Establish the trivial circular doubly linked list for B_c. Establishing this circular doubly linked list takes O(1) time.
  - Establish v as the pointer vertex of B_c. For now, assign the pointer values of $P(v_k)$. Establishing this pointer vertex takes O(1) time.
  - Remove vertices in X from the circular doubly linked iist of vertices in B_i to produce the circular doubly linked list of vertices in B_a and B_b. While doing so, make note of one vertex q_a from B_a. Producing these circular doubly linked lists takes O($|X|$) time.
  - If $P(B_i) \in X$, then we establish q_a as the new pointer vertex for B_a. Otherwise, if $P(B_i) \notin X$, then we establish v_1 new pointer vertex for B_b. Let q_a and q_b be the resulting pointer vertices of B_a and B_b, respectively. For now, assign the pointer values of $P(v_1)$ to q_a and q_b. Establishing these new pointer vertex takes O($|B_i|$) $\in$ O($|X|$) time.
  - Providing $I_L(B_i) \neq$ NIL [$I_L(v_i) \neq$ NIL], set $I_R(I_L(B_i))$ to B_a [$I_R(I_L(q_a))$ to q_a]. This assignment takes O(1) time.

    Once the above I_L pointer has been assigned, set $I_R(B_a)$ to B_b [$I_R(q_a)$ to q_b], and $I_L(B_b)$ to B_a [$I_L(q_b)$ to q_a]. As well, if $B_i = B_k$ [$b(v_i) = b(v_k)$], then set $I_R(B_b)$ to B_c [$I_R(q_b)$ to v] and $I_L(B_c)$ to B_b [$I_L(v)$ to q_b]. Otherwise, set $I_L(B_c)$ to B_k [$I_L(v)$ to v_k]. These assignments also take O(1) time.
  - Set $F_L(B_c)$ to B_b [$F_L(v)$ to q_b] and $F_R(B_c)$ to B'_j [$F_R(v)$ to v'_j]. These assignments take O(1) time.
  - For each block B in $\{F_L(B_a), \ldots, B_a\}$, if $F_R(B) = B_i$ [$f_R(P(B)) = b(v_i)$, then set $F_R(B)$ to B_b [$F_L(P(B))$ to q_b]. These assignments can be done in O($|X|$) time by following I_R pointers.

- For each block B in $\{B_b, \ldots, I_L(B_c)\}$, set $F_R(B)$ to B_c $[F_R(P(B))$ to $v]$. These assignments can be done in $O(|X|)$ time by following I_R pointers.
- For each block B in $\{B_a, \ldots, I_L(B_c)\}$, if $F_L(B) = B_i$ $[f_L(P(B)) = b(v_i)$, then set $F_L(B)$ to B_a $[F_L(P(B))$ to $q_a]$. These assignments can be done in $O(|X|)$ time by following I_R pointers.
- For each block B in $\{I_R(B_c), \ldots, B'_j\}$, set $F_L(B)$ to B_c $[F_L(P(B))$ to $v]$. These assignments can be done in $O(|X|)$ time by following I_R pointers.
- Set $s(B_c)$ $[s(v)]$ to one, $s(B_b)$ $[s(q_b)]$ to $|B_i \cap X|$, and $s(B_a)$ to $s(B_i) - |B_i \cap X|$ [subtract $s(q_b)$ from $s(q_a)$]. These assignments take $O(1)$ time.
- Relabel the blocks and adjust the b, f_L, and f_R values. This operation can be done in $O(n)$ time.

C.4.3 Deleting an edge

Let uv be the edge to be deleted, where X_u and X_v denote the neighbourhoods of u and v, respectively, in G. As well, let B_i and B_j be the blocks containing u and v, respectively, in the contig $B_1 \prec \ldots \prec B_k$ of the component C containing uv. Without loss of generality, let $1 \leq i \leq j \leq k$.

If $i = j$, then $i = j = k = 1$ and there are two case to be considered.

- B_1 contains another vertex besides u and v. We partition the contig B_1 to create a new contig $B_a \prec B_b \prec B_c$, where $B_a = \{u\}$, $B_b = B_1 \setminus \{u, v\}$, and $B_c = \{v\}$. Specifically, we amend the labelling as follows.
    - If u or v is the pointer vertex of B_1 $[P(v) \in \{u, v\}]$, then establish a new pointer vertex for B_1, with pointer values identical to those of $P(B_1)$. Specifically, use whichever of $nx(v)$ or $prev(v)$ is not u. This reassignment takes $O(|B(u, v)|) \in O(|X_u|) = O(|X_v|)$ time. Let q be the resulting pointer vertex of B_1.
    - Remove u and v from the circular doubly linked list of vertices in B_1 to produce the circular doubly linked list of vertices in B_b. This removal takes $O(1)$ time.
    - Establish the trivial circular doubly linked lists for B_a and B_c. These circular doubly linked lists can be created in $O(1)$ time.
    - Establish u as the pointer vertex of B_a and v as the pointer vertex of B_c. For now assign the pointer values of q to u and v. These pointer vertices can be established in $O(1)$ time.
    - Providing $I_L(B_1) \neq \text{NIL}$ $[I_L(q) \neq \text{NIL}]$, set $I_R(I_L(B_1))$ to B_a $[I_R(I_L(q))$ to $u]$. Similarly, providing $I_R(B_1) \neq \text{NIL}$, set $I_L(I_R(B_1))$ to B_c $[I_L(I_R(q))$ to $v]$. These assignments take $O(1)$ time.

Once the above I_L and I_R pointers have been set, set $I_R(B_a)$ to B_b $[I_R(u)$ to $q]$, $I_L(B_b)$ to B_a $[I_L(q)$ to $u]$, $I_R(B_b)$ to B_c $[I_R(q)$ to $v]$, and $I_L(B_c)$ to B_b $[I_L(v)$ to $q]$. These assignments also take O(1) time.

- Set $F_L(B_a)$ to B_a $[F_L(u)$ to $u]$, $F_R(B_a)$ to B_b $[F_R(u)$ to $q]$, $F_L(B_b)$ to B_a $[F_L(q)$ to $u]$, $F_R(B_b)$ to B_c $[F_R(q)$ to $v]$, $F_L(B_c)$ to B_b $[F_L(v)$ to $q]$, and $F_R(B_c)$ to B_c $[F_R(v)$ to $v]$. These assignments take O(1) time.
- Set $s(B_a)$ $[s(u)]$ and $s(B_c)$ $[s(v)]$ to one, and set $s(B_b)$ to $s(B_1) - 2$ [subtract two from $s(q)$]. These assignments take O(1) time.
- Relabel the blocks and adjust the b, f_L, and f_R values. This operation can be done in O(n) time.

- B_1 contains only u and v. We partition the contig B_1 to create two new contigs B_a and B_c, where $B_a = \{u\}$, $B_c = \{v\}$, and $B_a \prec B_c$. Without loss of generality, let us assume that v was the pointer vertex of B_1. We change the labelling as follows.
  - Establish the trivial circular doubly linked lists for B_a and B_c. Establishing these circular doubly linked lists takes O(1) time.
  - Establish u as the pointer vertex of B_a. For now, assign the pointer values of v to u. Establishing this pointer vertex takes O(1) time.
  - Providing $I_L(B_1) \neq$ NIL $[I_L(v) \neq$ NIL$]$, set $I_R(I_L(B_1))$ to B_a $[I_R(I_L(v))$ to $u]$. Similarly, providing $I_R(B_1) \neq$ NIL, set $I_L(I_R(B_1))$ to B_c $[I_L(I_R(v))$ to $v]$. These assignments take O(1) time.

    Once the above I_L and I_R pointers have been set, set $I_R(B_a)$ to B_c $[I_R(u)$ to $v]$, $I_L(B_c)$ to B_a $[I_L(v)$ to $u]$. These assignments also take O(1) time.
  - Set $F_L(B_a)$ to B_a $[F_L(u)$ to $u]$, $F_R(B_a)$ to B_a $[F_R(u)$ to $u]$, $F_L(B_c)$ to B_c $[F_L(v)$ to $v]$, and $F_R(B_c)$ to B_c $[F_R(v)$ to $v]$. These assignments take O(1) time.
  - Set $s(B_a)$ $[s(u)]$ and $s(B_c)$ $[s(v)]$ to one. These assignments take O(1) time.
  - Relabel the blocks and adjust the b, f_L, and f_R values. This operation can be done in O(n) time.

Now let us consider when $i \neq j$. Observe that if $1 < i$ $[f_L(u) \neq b(u)]$, $B_j = \{v\}$ $[nx(v) = v]$, $F_L(B_{i-1}) = F_L(B_i)$ $[f_L(I_L(u)) = f_L(u)]$, and $F_R(B_{i-1}) = B_{j-1}$ $[f_R(I_L(u)) = b(I_L(v))]$, then we must move u into B_{i-1}. Similarly, if $j < k$, $B_i = \{u\}$, $F_R(B_{j+1}) = F_R(B_j)$, and $F_L(B_{j+1}) = B_{i+1}$, then we must move v into B_{j+1}.

Exactly how the labelling is changed depends on whether u is moved into B_{i-1}, v is moved into B_{j+1}, $B_i = \{u\}$, and $B_j = \{v\}$. We consider each case, with respect to u, separately, noting that the same considerations must also be given for v.

- If u is to be moved into B_{i-1} and $B_i = \{u\}$ $[nx(u) = u]$, then we merge B_i into B_{i-1}. As we saw earlier when adding a vertex, such a merge takes O(1) time.
- If u is to be moved into B_{i-1} but B_i contains vertices other than u $[nx(u) \neq u]$, then we perform the following.
  - If u is the pointer vertex of B_i $[P(u) = u]$, then establish $nx(u)$ as the new pointer vertex for B_i, with pointer values identical to those of u. This reassignment takes $O(|B_i|) \in O(|X_u|)$ time. Let q be the resulting pointer vertex of B_i.
  - Remove u from the circular doubly linked list of the vertices in B_i and add u to the circular doubly linked list of the vertices in B_{i-1}, using $P(B_{i-1})$ $[P(I_L(u))]$ as a reference point. This move takes O(1) time.
  - Change the label of u to reflect that $P(B_{i-1})$ $[P(I_L(q))]$ is the new pointer vertex of its block. This change takes O(1) time.
  - For each B in $\{F_L(B_i), \ldots, B_{i-1}\}$, if $F_R(P(B)) = u$ then set $F_L(P(B))$ to q. These assignments can be done in $O(deg(B_i)) \in O(|X_u|)$ time, by recursively following I_L pointers to determine all such blocks B.
  - Decrease the value of $s(B_i)$ $[s(q)]$ by one and increase the value of $s(B_{i-1})$ $[s(P(u))]$ by one. These adjustments take O(1) time.
- If u was not moved into B_{i-1} and $B_i = \{u\}$, then we need do nothing yet.
- If u was not moved into B_{i-1} and B_i contains vertices other than u, then we must partition B_i into $B_a \prec B_b$, where $B_a = \{u\}$ and $B_b = B_i \setminus \{u\}$ (in the case of v, we would partition B_j into $B_a \prec B_b$, where $B_a = B_j \setminus \{v\}$ and $B_b = \{v\}$). Specifically, the labelling changes as follows.
  - If u is the pointer vertex of B_i $[P(u) = u]$, then establish $nx(u)$ as the new pointer vertex for B_i, with pointer values identical to those of u. This reassignment takes $O(|B(u)|) \in O(|X_u|)$ time. Let q be the resulting pointer vertex of B_i.
  - Remove u from the circular doubly linked list of vertices in B_i to produce the circular doubly linked list of vertices in B_b. This removal takes O(1) time.
  - Establish the trivial circular doubly linked list for B_a. Establishing this circular doubly linked list takes O(1) time.
  - Establish u as the pointer vertex of B_a. For now, assign the pointer values of q to u. Establishing this pointer vertex takes O(1) time.
  - Providing $I_L(B_i) \neq$ NIL $[I_L(q) \neq$ NIL$]$, set $I_R(I_L(B_i))$ to B_a $[I_R(I_L(q))$ to $u]$. As well, set $I_L(I_R(B_i))$ to B_b $[I_L(I_R(q))$ to $q]$. These assignments take O(1) time.

Once these I_L and I_R pointers have been assigned, set $I_R(B_a)$ to B_b [$I_R(u)$ to q] and $I_L(B_b)$ to B_a [$I_L(q)$ to u]. These assignments also take O(1) time.

  - For each block B in $\{F_L(B_i), \ldots, B_{i-1}\}$, if $F_R(P(B)) = u$, then set $F_R(P(B))$ to q. As well, for each block B in $\{B_{i+1}, \ldots, B_j\}$, if $P(F_L(P(B))) = q$ then set $F_L(B)$ to B_a [$F_L(P(B))$ to u]. These assignments can be done in $O(deg(B_i)) \in O(|X_u|)$ time, by recursively following I_L and I_R pointers to determine all such blocks B.
  - Set $s(B_a)$ [$s(u)$] to one and $S(B_b)$ to $s(B_i) - 1$ [subtract one from $s(q)$]. This takes O(1) time.

Once the above actions involving u and v have been completed, we set $F_R(B(u))$ to $I_L(B(v))$ [$F_R(u)$ to $I_L(v)$] and $F_L(B(v))$ to $I_R(B(u))$ [$F_L(v)$ to $I_R(u)$]. These pointers can be set in O(1) time. We then relabel the blocks and adjust the b, f_L, and f_R values. These values can be adjusted in $O(n)$ time.

C.4.4 Adding an edge

Let uv be the edge to be added, where X_u and X_v denote the neighbourhoods of u and v, respectively, in G. Without loss of generality, let us assume that $b(u) < b(v)$. While considering the addition of a vertex, we saw that, by traversing F_R pointers beginning at u, we can determine, in $O(n)$ time, whether u and v belong to the same component.

We consider the following cases.

1. The vertices u and v belong to distinct components. In this case we will need to know information about all the blocks in the components containing u and v. Specifically, we determine all the blocks by following F_L and F_R pointers, keeping a reference vertex v_i from each block B_i. Gathering this information can take as much as $\Theta(n)$ time.

   Let $\Phi = B_1 \prec \ldots \prec B_k$ be the contig of the component containing u, and let $\Psi = B'_1 \prec \ldots \prec B'_l$ be the contig of the component containing v. To demonstrate the action of the relabeller, we consider the scenario in which $u \in B_k$ and $v \in B'_1$, with $B'_1 = \{v\}$ but not $B_k \neq \{u\}$. Let B_a be the split block $B_k \setminus \{u\}$, and B_b be the block $\{u\}$. Our labelling changes as follows.

   - If $I_R(B_k) \neq B'_1$ [$b(I_R(v_k)) = b(v'_1)$], then move Φ over to Ψ.
     - Set $I_R(I_L(B'_1))$ to $I_R(B'_l)$ [$I_R(I_L(v'_1))$ to $I_R(v'_l)$] and $I_R(B'_l)$ to $I_R(B_k)$ [$I_R(v'_l)$ to $I_R(v_k)$]. As well, providing $I_R(B'_l) \neq$ NIL [$I_R(v'_l) \neq$ NIL], set $I_L(I_R(B'_l))$ to $I_L(B'_1)$ [$I_L(I_R(v'_l))$ to $I_L(v'_1)$]. These assignments can be made in O(1) time. Once the above I_L and I_R pointers have been assigned, set $I_R(B_k)$ to B'_1 [$I_R(v_k)$ to v'_1], and $I_L(B'_1)$ to B_k [$I_L(v'_1)$ to v_k]. These assignments also take O(1) time.

    - Relabel the blocks and adjust the b, f_L, and f_R values. This operation can be done in $O(n)$ time.
- Split B_k to reflect the straight enumeration of G'.
    - If u is the pointer vertex of B_k $[P(u) = u]$, then establish $nx(u)$ as the new pointer vertex for B_a. This reassignment takes $O(|B(u)|) \in O(|X_u|)$ time. Let q_a be the resulting pointer vertex of B_a.
    - Remove u from the circular doubly linked list of vertices in B_k to produce the circular doubly linked list of vertices in B_a and B_b. Producing these circular doubly linked lists takes $O(|B_k|) \in O(|X_u|)$ time.
    - Establish u as the pointer vertex of B_b and give its pointers the same values as q_a. Establishing this pointer vertex takes $O(1)$ time.
    - Providing $I_L(B_k) \neq$ NIL $[I_L(q_a) \neq$ NIL$]$, set $I_R(I_L(B_k))$ to B_a $[I_R(I_L(q_a))$ to $q_a]$. This assignment takes $O(1)$ time.

      Once the above I_L and I_R pointers have been assigned, set $I_R(B_a)$ to B_b $[I_R(q_a)$ to $u]$, $I_L(B_b)$ to B_a $[I_L(u)$ to $q_a]$, and $I_L(B'_1)$ to B_b $[I_L(v)$ to $u]$. These assignments also take $O(1)$ time.
    - Set $F_R(B_b)$ to B'_1 $[F_R(u)$ to $v]$, and $F_L(B'_1)$ to B_b $[F_L(v)$ to $u]$. These assignments take $O(1)$ time.
    - For each block B in $\{F_L(B_a), \ldots, B_a\}$, set $F_R(B)$ to B_b $[F_R(P(B))$ to $u]$. These assignments can be done in $O(|X_u|)$ time by following I_L pointers.
    - Set $s(B_b)$ $[s(u)]$ to one, and subtract one from the value of $s(B_a)$ $[s(q_a)]$. These assignments take $O(1)$ time.
    - Relabel the blocks and adjust the b, f_L, and f_R values. This operation can be done in $O(n)$ time.

2. The vertices u and v belong to the same component. Let $B_1 \prec \ldots \prec B_k$ be the contig of the component containing u and v, where $u \in B_i$ and $v \in B_j$, for some $1 \leq i < j \leq k$. We consider two further cases.

    (a) B_i and B_j are end blocks $[f_L(u) = b(u)$ and $f_R(v) = b(v)]$. In this case, $X_u = X_v$. By Lemma 6.7 (umbrella property), the contig contains three blocks, namely, $\{u\} \prec X_u \prec \{v\}$. The new component will consist of a single block, formed by merging the three blocks into one new block B_a.

    Letting q be the pointer vertex of the block X_u, the labelling changes as follows.

    - Providing $I_L(\{u\}) \neq$ NIL $[I_L(u) \neq$ NIL$]$, set $I_R(I_L(\{u\}))$ to B_a $[I_R(I_L(u))$ to $q]$. Similarly, providing $I_R(\{v\}) \neq$ NIL $[I_R(v) \neq$ NIL$]$, set $I_L(I_R(\{v\}))$ to B_a $[I_L(I_R(v))$ to $q_2]$. These assignments take $O(1)$ time.

- Establish the circular linked list for B_a by adding u and v to the circular doubly linked list of vertices in X_u, using q as a reference. Moreover, change the labels of u and v to reflect that they are no longer pointer vertices, and q is the pointer vertex of their new block. These changes take O(1) time.
- Set the value of $s(B_a)$ to $s(X_u) + 2$ [add two to the value of $s(q)$]. This addition takes O(1) time.
- Relabel the blocks and adjust the b, f_L, and f_R values. This operation can be done in O(n) time.

(b) At least one of B_i and B_j is not an end block $[f_L(u) \neq b(u)$ or $f_R(v) \neq b(v)]$. In this case, $X_u \neq X_v$. If $B_i = \{u\}$ $[nx(u) = u]$, $F_R(B_{j-1}) = F_R(B_j)$ $[f_R(I_L(v)) = f_R(v)]$ and $F_L(B_{j-1}) = B_i$ $[f_L(I_L(v)) = b(u)]$, then we move v from B_j into B_{j-1}. Similarly, if $B_j = \{v\}$ $[nx(v) = v]$, $F_L(B_{i+1}) = F_L(B_i)$ $[f_L(I_R(u)) = f_L(u)]$ and $F_R(B_{i+1}) = B_j$ $[f_R(I_R(u)) = b(v)]$, then we move u from B_i into B_{i+1}. This moving of u and v is virtually identical to one of the cases discussed when considering the deletion of a edge.

Our labelling is modified according to whether or not u and v are moved.

- If u is to be moved into B_{i+1} and $B_i = \{u\}$ $[nx(u) = u]$, then we merge B_i into B_{i+1}. As we saw earlier when adding a vertex, such a merge takes O(1) time.
- If u is to be moved into B_{i+1} but B_i contains vertices other than u $[nx(u) \neq u]$, then we perform the following.
  - If u is the pointer vertex of B_i $[P(u) = u]$, then establish $nx(u)$ as the new pointer vertex for B_i, with pointer values identical to those of u. This reassignment takes $O(|B_i|) \in O(|X_u|)$ time. Let q be the resulting pointer vertex of B_i.
  - Remove u from the circular doubly linked list of the vertices in B_i and add u to the circular doubly linked list of the vertices in B_{i+1}, using $P(B_{i+1})$ $[P(I_R(u))]$ as a reference point. This move takes O(1) time.
  - Change the label of u to reflect that $P(B_{i+1})$ $[P(I_R(q))]$ is the new pointer vertex of its block. This change takes O(1) time.
  - For each B in $\{B_{i+1}, \ldots, F_R(B_i)\}$, if $F_L(P(B)) = u$, then set $F_R(P(B))$ to q. These assignments can be done in $O(deg(B_i)) \in O(|X_u|)$ time by recursively following I_R pointers to determine all such blocks B.
  - Decrease the value of $s(B_i)$ $[s(q)]$ by one and increase the value of $s(B_{i+1})$ $[s(P(u))]$ by one. These adjustments take O(1) time.
- If u was not moved into B_{i+1} and $B_i = \{u\}$, then we need do nothing yet.

- If u was not moved into B_{i+1} and B_i contains vertices other than u, then we must partition B_i into $B_a \prec B_b$, where $B_a = B_i \setminus \{u\}$ and $B_b = \{u\}$ (in the case of v, we would partition B_j into $B_a \prec B_b$, where $B_a = \{v\}$ and $B_b = B_j \setminus \{v\}$). Specifically, the labelling changes as follows.
  - If u is the pointer vertex of B_i $[P(u) = u]$, then establish $nx(u)$ as the new pointer vertex for B_i, with pointer values identical to those of u. This reassignment takes $O(|B(u)|) \in O(|X_u|)$ time. Let q be the resulting pointer vertex of B_i.
  - Remove u from the circular doubly linked list of vertices in B_i to produce the circular doubly linked list of vertices in B_a. This removal takes $O(1)$ time.
  - Establish the trivial circular doubly linked list for B_b. Establishing this circular doubly linked list takes $O(1)$ time.
  - Establish u as the pointer vertex of B_a. For now, assign the pointer values of q to u. Establishing this pointer vertex takes $O(1)$ time.
  - Providing $I_L(B_i) \neq \text{NIL}$ $[I_L(q) \neq \text{NIL}]$, set $I_R(I_L(B_i))$ to B_a $[I_R(I_L(q))$ to $q]$. As well, set $I_L(I_R(B_i))$ to B_b $[I_L(I_R(q))$ to $u]$. These assignments take $O(1)$ time.

    Once these I_L and I_R pointers have been assigned, set $I_R(B_a)$ to B_b $[I_R(q)$ to $u]$, and $I_L(B_b)$ to B_a $[I_L(u)$ to $q]$. These assignments also take $O(1)$ time.
  - For each block B in $\{F_L(B_i), \ldots, B_i\}$, if $P(F_R(P(B))) = q$, then set $F_R(P(B))$ to u. As well, for each block B in $\{B_{i+1}, \ldots, B_{j-1}\}$, if $F_L(P(B)) = u$ then set $F_L(B)$ to B_a $[F_L(P(B))$ to $q]$. These assignments take $O(|X_u|)$ time.
  - Set $s(B_b)$ $[s(u)]$ to one and $S(B_a)$ to $s(B_i) - 1$ [subtract one from $s(q)$]. This takes $O(1)$ time.

Once the above actions involving u and v have been completed, we set $F_R(B(u))$ to $B(v)$ $[F_R(u)$ to $v]$ and $F_L(B(v))$ to $B(u)$ $[F_L(v)$ to $u]$. These pointers can be set in $O(1)$ time. We then relabel the blocks and adjust the b, f_L, and f_R values. These values can be adjusted in $O(n)$ time.

Index

adjacency labelling scheme, **3**, *see* implicit representation
  almost trees(k), 14
  arboricity-k graphs, 3, 14, 15
  asteroidal triple free graphs, 14
  autographs, 14
  balanced, 12, 14, 15
  bandwidth-k graphs, 14
  bipartite graphs, 12, 14, 15
  boxicity-k graphs, 14, 15
  chain graphs, 14
  chordal comparability graphs, 14
  chordal graphs, 12, 14
  circle graphs, 15
  circular arc graphs, 15
  cobipartite graphs, 14, 15
  cographs, 15
  comparability graphs, 15
  containment graphs, 15
  convex bipartite graphs, 15
  cycles, 15
  k-decomposable graphs, 15
  degree-k graphs, 14, 15
  disk intersection graphs, 15
  k-dot product graphs, 13, 15
  EPT graphs, 15
  forests, 15
  C_3-free graphs, 14
  C_3,C_4-free graphs, 14
  C_3,$K_{1,3}$-free graphs, 14
  $K_{1,3}$-free graphs, 14
  $K_{3,3}$-free graphs, 14
  K_5-free graphs, 14
  P_4-free graphs, 14
  from adjacency lists, 6
  from adjacency matrix, 6, 12, 17, 54
  general graphs, 14
  genus-k graphs, 15
  hereditary degree-k graphs, 15
  interval graphs, 3, 11, 14, 15
  k-interval graphs, 15
  line graphs, 3, 8–9, 15
  outdegree-k graphs, 9–10, 14, 15
  outerplanar graphs, 13, 15
  permutation graphs, 15
  planar graphs, 15
  posets of dimension-k, 15
  proper interval graphs, 14, 15
  quality, 10–12
  space-optimal, 11
  k-sparse graphs, 12–13, 15
  split graphs, 14, 15
  strongly space-optimal, 11, 12, 16
  threshold graphs, 15
  threshold tolerance graphs, 15
  total graphs, 15
  transitive closures of rooted trees, 7–8, 15
  trees, 7, 10, 11, 15
  uniformly k-sparse graphs, 15
adjacency list, 2, 6
adjacency matrix, 2, 6
adjacent, **1**
almost tree(k), 14, **118**
arboricity, **118**
arboricity-k graph, 3, 4, 14, 15, 32, 33
asteroidal triple, **118**
asteroidal triple free graph, 14
astral triple, 82, 83, **118**
autograph, 14, **118**

bandwidth, **118**
bandwidth-k graph, 14
base graph, *see* line graph, base
biclique, **54**, **118**
  maximal, 4, 5, 34, 54, 68, 81, 112, 118
r-bic, **5**, **34**, **54**, 55, 67–81, **81**, **112**, **118**
binary tree, 21, **118**
bipartite graph, 12, 14, 15, 21, 119
block, 83–84
boxicity, **119**
boxicity-k graph, 14, 15
broadcast protocols, 20

chain graph, 14, **119**
chordal bipartite graph, 12, **119**
chordal comparability graph, 14
chordal graph, 12, 14, **119**
circle graph, 15, **119**
circular arc graph, 15, 21, **119**
circular doubly linked list, 39, 55, 68, 86
circular linked list, 4, 111
claw, **119**
clique
  maximal, 5, 34, 54, 56, 81, 112, 121
cliquewidth, **119**
cliquewidth-k graph, 21, **119**
closed neighbourhood, **1**
co-class, **10**, 14
cobipartite graph, 14, 15, **119**
cograph, 15, **119**
comparability graph, 15, **119**
component, **2**
computation model
  log-cost RAM, *see* log-cost RAM
  unit-cost RAM, *see* unit-cost RAM
  word-level RAM, *see* word-level RAM
connected graph, **2**

- containment class, 18, **120**
- containment graph, 15, **120**
- contig, **84**, 85
- convex bipartite graph, 15, **120**

- decoder, **3**, 7–12, 16, 19, 39, 56, 68, 86, 124
- *k*-decomposable graph, 15
- *k*-decomposable, **120**
- degree, **1**
- degree-*k* graph, 14, 15, 21
- degree-*k* planar graph, 21
- degree-3 graph, 21
- directed graph, **1**
- disk intersection graph, 15, **120**
- distance hereditary graph, 21, **120**
- distinct graphs
  - labelled, **2**
  - unlabelled, **2**
- domino, **54**, 56
- *k*-dot product graph, **13**, 13, 15, **120**
- dynamic adjacency labelling scheme, *see* dynamic labelling scheme, adjacency
- dynamic informative labelling scheme, *see* dynamic labelling scheme
- dynamic labelling scheme, 22–34, 112, 113
  - adjacency, **23**
    - arboricity-*k* graphs, 4, 32, 33
    - *r*-bics, 5, 34, 55, 67–81, 112
    - line graph, 34
    - line graphs, 5, 38–52, 111
    - *r*-minoes, 5, 34, 81, 112
    - *r*-minoes, 55–67
    - proper interval graphs, 5, 82–107, 112
    - trees, 24
  - ancestor
    - rooted trees, 4, 33
  - assumptions, 24–25, 57, 69
  - distance
    - trees, 4, 33
    - weighted cycles, 34
    - weighted trees, 34, 112
  - flow
    - weighted trees, 112
  - informative, **26**
  - modification excess, *see* modification excess
  - modification locality, *see* modification locality
  - quality, 4, 26–29, 111
  - routing
    - weighted trees, 112
  - separation level
    - weighted trees, 112

- edge, **1**
- edge set, **1**
- end pointer, 85, 87
- EPT graph, 15, **120**
- error-detection, 4, **23**, **26**, 29–32, 63, 75, 111

- far pointer, 85
- finite graph, **1**
- forest, 15, 22, **120**
- C_3-free graph, 14
- C_3,C_4-free graph, 14
- C_3,$K_{1,3}$-free graph, 14
- H-free, **118**
- $K_{1,3}$-free, 14
- $K_{3,3}$-free graph, 14
- K_5-free graph, 14
- P_4-free graph, 14

- genus, **120**
- genus-*k* graph, 15
- graph, **1**
- graph recognition, 29–32

- hereditary degree-*k* graph, 15, **120**
- hereditary property, **120**
- hypercube, 21, **120**
- hypergraph, 17, **121**
  - line graph of, **17**, **121**
  - rank, **17**, **121**

- identifier, 4, 6–10, 24, 39, 56, 68, 86
- implicit representation, **16**, 16–18
- incident, **1**
- informative labelling scheme, 3, **19**, 18–19, 125
  - adjacency, *see* adjacency labelling scheme
  - ancestor, 3, 20
    - rooted trees, 19, 21
  - applications, 19–20
  - bounded distance
    - trees, 21
  - center of three vertices, 3
    - trees, 21
  - distance, 3, 20
    - binary trees, 21
    - bipartite graphs, 21
    - circular arc graphs, 21
    - cliquewidth-*k* graphs, 21
    - cycles, 21
    - degree-*k* graphs, 21
    - degree-*k* planar graphs, 21
    - degree-3 graphs, 21
    - distance hereditary graphs, 21
    - general graphs, 21
    - hypercubes, 21
    - interval graphs, 21
    - meshes, 21
    - permutation graphs, 21
    - planar graphs, 21
    - proper interval graphs, 21
    - recursive $r(n)$-separator graphs, 21
    - tori, 21
    - trees, 21
    - treewidth-*k* graphs, 21
    - weighted binary trees, 21
    - weighted c-decomposable graphs, 21
    - weighted k-outerplanar graphs, 21
    - weighted series parallel graphs, 21
    - weighted trees, 21
    - well (α, g)-separated graphs, 21
  - edge-connectivity
    - general graphs, 21

  - flow, 20
    - general graphs, 22
  - nearest common ancestor, 3, 20
    - rooted trees, 22
  - parent, 20
  - quality, 19
  - reachability
    - planar digraphs, 22
  - routing, 3, 20
    - forests, 22
    - trees, 22
  - separation level
    - rooted trees, 22
  - sibling, 20
  - Steiner tree
    - weighted graphs, 22
  - k-vertex connectivity
    - general graphs, 22
- intersection class, 17, 35, **121**
- intersection graph, **121**
- intersection number, 17
- interval graph, 3, 11, 14, 15, 21, 83, 112, **121**
- interval number, **121**
- k-interval graph, 15, **121**
- isomorphic graphs, **2**

- labelled graph, **2**
- line graph, 3, 5, **8**, 8–9, 15, 34, **35**, 38–52, 111, **121**, *see* hypergraph, line graph of
  - base, **8**, 35, 39
- log-cost RAM, 124
- loop, **1**

- marker, **3**, 7–12, 16, 19, 63, 82, 87, 124
- mesh, 21, **121**
- r-mino, **5**, **34**, **54**, 55–67, **81**, **112**, **121**
- modification excess, **29**, 27–29, 43, 46, 52, 60, 63
- modification locality, **29**, 27–29, 43, 46, 52, 60, 63, 65, 67

- near pointer, 85
- neighbourhood, **1**
  - of a block, 83

- open neighbourhood, **1**
- outdegree-k graph, **9**, 9–10, 14, 15, **121**
- outerplanar graph, 13, 15, **121**
- k-outerplanar, **121**
- outneighbour, 9
- overlap class, 18

- partial order, **121**
- partition isomorphism, **36**, 35–38, 41
- path, **2**
- permutation graph, 15, 21, **122**
- persistent labels, 4, 33
- planar digraph, 22
- planar graph, 15, 21, **122**
- pointer vertex, **85**, 85–87
- poset, 15, **122**
- prefix-free binary string, 19
- proper interval graph, 5, 14, 15, 21, **82**, 82–107, 112, **122**

- reciprocal pointer, 56
- recognition, 63, 75, 88
- recursive $r(n)$-separator graph, 21, **122**
- relabeller, **23**, 26, 27, 41–52, 57–67, 70–81, 87–107, 124
- rooted tree, 4, 19, 21, 22, 33, **122**
- routing algorithms, 20

- self pointer, 85, 87
- series parallel graph, **122**
- simple graph, **1**
- k-sparse graph, **12**, 12–13, 15, **122**
- split graph, 14, 15, **122**
- straight enumeration, **84**, 84, 112
- subclass, 10, 14
- superclass, 10, 14

- threshold graph, 15, **122**
- threshold tolerance graph, 15, **122**
- torus, 21, **122**
- total graph, 15, **122**
- transitive closure of rooted tree, 7–8, 15, **123**
- tree, 4, 7, 10, 11, 15, 21, 22, 24, 33
- treewidth, **123**
- treewidth-k graph, 21
- triangle, **123**

- undirected graph, **1**
- uniformly k-sparse graph, 15, **123**
- unit interval graph, **123**
- unit-cost RAM, 124
- universal graph, 18, **123**
- unlabelled graph, **2**

- valid set, **44**
- vertex, **1**
- vertex set, **1**

- walk, **2**
- weak linear order, 92
- weighted binary tree, 21
- weighted cycle, 34
- weighted c-decomposable graph, 21
- weighted graph, 22
- weighted k-outerplanar graph, 21
- weighted series parallel graph, 21
- weighted tree, 21, 34, 112
- well (α, g)-separated graph, 21, **123**
- word-level RAM, 6, 11, 124

- XML search engine, 20